UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VALÉRIA SOLDERA GIRELLI

A Study of the Prefetcher Impact on High-Performance Computing Applications

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Philippe O. A. Navaux
Coadvisor: Dr. Francis Birck Moreira

Porto Alegre
June 2021

ACKNOWLEDGEMENTS

I would like to thank my parents for always supporting me and teaching me since my childhood that I could do anything I wanted. Thanks to my mom, Lurdes, for always showing me so much strength and always knowing exactly what I needed to hear when things became difficult. Thanks to my dad, Gilmar, for teaching me how to be patient and hardworking. I cannot thank my friends enough for all the support and understanding during all these years.

I would like to give a special thanks to those whose contributions were essential to the development of this work: to my advisor, Prof. Navaux, for all his support, guidance, and for having always shown me the right directions; to my co-advisor, Francis, for being fundamental in my growth and for being by my side during the most significant moments of this path; to all my friends at GPPD for every tiny thing you taught me.

I wish I could name here everyone who has been indispensable through every step I gave, but you know me, and you know how incredibly grateful I am for everything you have done for me.

ABSTRACT

Data prefetching algorithms are widely used in modern processors as a tool to mitigate the higher latency of memory accesses with respect to processor latency to execute instructions. However, understanding the contribution of prefetching to the application performance is a difficult task when we consider the high complexity found in the several architectures and prefetchers available. Developing accurate architecture simulators is also a challenge, especially when considering High-Performance Computing systems (HPC) with several processor cores. In this work, we contribute to shed light on the role of data prefetchers in the performance of parallel HPC applications, considering both the prefetcher algorithms offered in the real hardware and in the simulators. We performed a careful experimental investigation, executing the NAS parallel benchmark (NPB) on a real Skylake machine and in a simulated environment with the ZSim and Sniper simulators, using prefetcher algorithms offered by both Skylake and the simulators. Our experimental results show that: (i) prefetching from the L3 to L2 cache is responsible for the larger percentage of performance improvement, (ii) the memory contention in the parallel execution constrains the effectiveness of the prefetcher, (iii) the parallel memory contention in Skylake is poorly simulated by ZSim and Sniper, and (iv) the non-inclusive L3 cache present in the Skylake architecture hinders the accurate simulation of NPB with the Sniper prefetchers.

Keywords: Architecture Simulation. Computer Architecture. Parallel Architecture. Data Prefetching.

Um Estudo Sobre o Impacto de Prefetchers em Aplicações de Alto Desempenho

RESUMO

Algoritmos de *prefetching* são vastamente utilizados em processadores modernos como uma forma de mitigar a diferença de desempenho que existe entre o processador e o sistema de memória. No entanto, se considerarmos a complexidade das diversas arquiteturas de computadores e dos algoritmos de *prefetching* disponíveis, compreender como o *prefetcher* afeta o desempenho das aplicações se torna uma tarefa difícil. Além disso, desenvolver simuladores de arquiteturas que sejam precisos também é desafiador, e essa tarefa pode se tornar ainda mais difícil em um contexto no qual sistemas de computação de alto desempenho (*High-Performance Computing* – HPC) possuem dezenas de núcleos de processamento. Neste trabalho, nós buscamos ampliar o conhecimento a respeito do papel do sistema de *prefetching* sobre o desempenho de aplicações paralelas de alto desempenho, estudando tanto os algoritmos presentes em uma máquina real quanto os oferecidos por simuladores de arquiteturas. Em nossa investigação experimental, nós executamos o conjunto de *benchmarks* paralelos NPB (*NAS Parallel Benchmarks*) em uma máquina de arquitetura Skylake, bem como em um ambiente de simulação composto pelos simuladores de arquiteturas paralelas ZSim e Sniper. Nossos resultados mostram que: (i) realizar *prefetcher* da *cache* L3 para a *cache* L2 apresentou os melhores ganhos de desempenho, (ii) a contenção de memória observada durante a execução paralela acaba restringindo o efeito do *prefetcher*, (iii) ambos os simuladores ZSim e Sniper simulam de forma imprecisa a contenção de memória observada na máquina Skylake, e (iv) a característica não-inclusiva da cache L3 da Skylake dificulta a simulação do NPB com os algoritmos de *profetching* do Sniper.

Palavras-chave: Simulação de Arquitetura. Arquitetura de Computadores. Arquiteturas Paralelas. Prefetch de Dados.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ARM	Advanced RISC Machines
B	Bytes, a unit of digital information equal to 8 bits
CPU	Central Processing Unit
DCU	Data Cache Unit
DBT	Dynamic Binary Translation
DRAM	Dynamic Random-Access Memory
EDP	Energy-Delay Product
EmbPar	Embarassingly parallel
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GHB	Global History Buffer
GHz	Gigahertz, a unit of frequency equal to 1,000,000,000 (one billion) Hz (hertz)
GB	Gigabyte, a unit of information equal to 1,073,741,824 bytes
HPC	High-Performance Computing
I/O	Input/Output, communication between an information processing system
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
KB	Kilobyte, a unit of information equal to 1024 bytes
L1	First Level Cache
L1D	First Level Data Cache
L1I	First Level Instruction Cache
L2	Second Level Cache
L3	Third Level Cache, usually is the last level of cache

LLC	Last Level Cache
LRU	Least Recently Used
MB	Megabyte, a unit of information equal to 1,048,576 bytes
MLP	Memory Level Parallelism
MIPS	Microprocessor without Interlocked Pipelined Stages, a RISC instruction set architecture developed by MIPS Computer Systems
MSHR	Miss Status Holding Register
NPB	Numerical Aerodynamic Simulation Parallel Benchmark
PAPI	Performance Application Programming Interface
PMC	Performance Monitor Counter
RFO	Request for Ownership
ROB	Reorder Buffer
SRAM	Static Random-Access Memory
TLB	Translation Lookaside Buffer

CONTENTS

1 INTRODUCTION

In the last decades, there have been significant advances in the performance of processors, exemplified by the reduction of transistor size and the increase in the number of cores in a processor. Conversely, the memory subsystem did not advance as significantly as processors due to technological constraints that keep the memory from achieving a higher performance. This problem is referred in the literature as the memory wall (WULF; MCKEE, 1996) and it is the responsible for several architectural inefficiencies, such as the instruction execution stall (HENNESSY; PATTERSON, 2017) and the emergence of contention on shared resources (BAKHSHALIPOUR et al., 2019a). For this reason, there is a keen effort in computer architecture research to overcome these memory limitations. An example of a technology used to mitigate the memory latency is prefetching, a technique that identifies access patterns from each core, creates speculative memory requests, and fetches data that can be potentially useful to the cache prior to the request is actually made.

In High-Performance Computing (HPC) systems, many other problems associated with the parallel nature of applications and architectures may arise. Typically, HPC applications are parallel, mainly using shared memory for communication between the threads (HENNESSY; PATTERSON, 2017). When distinct threads access the same memory addresses, it becomes crucial for the correctness of the application to maintain data coherence in the cache levels of the several cores; that is, a given data cannot present different values depending on at which core it is found. Moreover, the action of the data coherence protocol results in a set of data transfers between cores and data invalidation, which may also unpredictably change the data path through the memory hierarchy. The complexity of the memory system in multi-core architectures along with the prefetching introduces a new level of complexity. The hindrance lies, therefore, in understanding how a prefetcher affects the processing performance of parallel HPC applications.

To further complicate the matter, in computer architecture research, physical implementation and analysis are infeasible due to the high complexity and manufacturing costs. Consequently, it became fundamental to develop a way of simulating computer architecture behavior. Architecture simulators with various detailing levels and approaches were proposed (AKRAM; SAWALHA, 2019), and simulation is now considered the primary mechanism to implement and evaluate new ideas (SKADRON et al., 2003), such as new prefetching algorithms. In the context of HPC systems, simulators also need to

consider the obstacles that arise with the inherent parallelism of such systems. multi-core architecture simulators that support parallel workloads and accurately simulate interaction between cores are crucial. However, given the complexity of multi-core systems and the difficulty of fully understanding parallel interactions, many gaps in parallel simulation still remain to be filled.

In this work, a state-of-the-art overview is presented, introducing information regarding the memory hierarchy behavior, distinct prefetching mechanisms, and architecture simulation approaches. We developed a careful experimental investigation, executing the Numerical Aerodynamic Simulation Parallel Benchmark (NPB) (JIN et al., 1999) in an Intel Skylake machine and in two broadly used architecture simulators, the ZSim (SANCHEZ; KOZYRAKIS, 2013) and Sniper (CARLSON; HEIRMANT; EECKHOUT, 2011) simulators. We aimed to shed light on the following questions: *(i) how does prefetching affect the processing performance of parallel, HPC applications?*, and *(ii) how accurately can state-of-the-art architecture simulators simulate HPC applications, with and without prefetchers?*

More specifically, this work presents the following set of contributions:

- We show experimental evidence that an L2 memory prefetcher is more efficient in comparison with an L1 prefetcher. Since the L3 access latency is higher than the L2 access latency, the L2 prefetcher avoids excessive L3 cache accesses and better contributes to performance when comparing to the L1 prefetcher; in fact, we demonstrate that the standalone L2 prefetcher can achieve similar performance gains compared to using both prefetchers enabled;
- We show evidence that the contribution the prefetchers has on the performance is limited by the level of parallelism of the application, mainly due to the increase in communication and memory contention as the level of parallelism increases;
- The Skylake and NPB simulation with ZSim and Sniper had poor accuracy in predicting the NPB applications performance, with and without simulating prefetcher algorithms, mainly due distinctions of the models and prefetcher algorithms present in ZSim and Sniper when compared to Skylake.

The remainder of this monograph is organized as follows: In Chapter 2 we delve into the background of this work, illustrating in details the memory system and the prefetcher system action, as well as the related works. In this Chapter, we also introduce computer architecture simulators and their possible classification categories. Chapter 3 embodies

the motivation behind the work proposed in this monograph, demonstrating some gaps that remain to be filled and are addressed by this work. In Chapter 4, we present the simulators applied for this work and describe the experimental campaign performed – such as the experimental setup and the benchmarks used. In Chapter 5 and 6, we present the experimental results, the main findings and observations obtained by our study regarding the real machine and the simulation environment, respectively. In Chapter 7 we propose a discussion regarding prefetching, good practices and guidelines, and in Chapter 8, we give our concluding remarks, and we discuss future research directions.

We follow a reproducible and open methodology in our investigation. A complementary material of this work is publicly available[1], containing the application code, the data analysis code, and all data collected during experiments that culminated in this monograph and in the paper "Investigating Memory Prefetcher Performance Over Parallel Applications: From Real to Simulated" (GIRELLI et al.,).

[1] <https://gitlab.com/msserpa/prefetcher-ccpe>

2 BACKGROUND

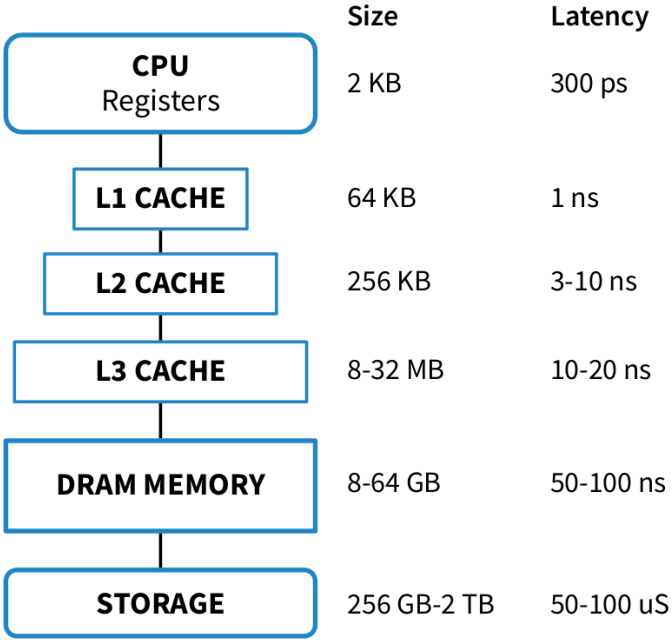
In the following sections, we delve into the memory system, detailing the technologies applied, the hierarchical organization, and the performance constraints involved. We also examine prefetching, its function in the memory hierarchy, and state-of-the-art prefetching algorithms. Moreover, we present the possible classifications and implementation techniques applied by computer architecture simulators while analyzing each class' main works.

2.1 The Memory Subsystem and The Prefetcher

In the organization of a computer, the memory available for a processor is distributed in different levels, in both on-chip and off-chip regions. One of the main distinctions between on-chip and off-chip memories is their access latencies, a result of the different technologies applied in their manufacturing process. The technology commonly used in memories found outside the processor is the DRAM (Dynamic Random-Access Memory). The DRAM, normally employed as the computer main memory, is organized in several banks, and each bank contains several rows and columns in a conceptually matrix fashion. In order to put more bits per chip, DRAM modules use only one capacitor to store a bit, which works together with a transistor that controls access to the capacitor. This capacitor is periodically "refreshed" in order to prevent data loss due to charge dissipation (HENNESSY; PATTERSON, 2017). To read the information from a DRAM row, its electrical values are loaded into a row buffer, destroying the information on the row and requiring the data to be written back after a read (HENNESSY; PATTERSON, 2017). This complex process of reading from a DRAM row and the need for restoring the bit-line values, storing back into the physical row, and periodically refreshing every row, are the main reasons why accessing data stored inside DRAM memories requires a higher time.

On the other side, the technology frequently used for on-chip memories is the SRAM (Static Random-Access Memory). SRAM memories, also known as cache memories, most commonly use six MOS transistors to store a bit, eliminating the need for refresh and preventing the data to be disturbed when read (HENNESSY; PATTERSON, 2017). These electrical differences, and especially the higher size and capacity of the DRAM physical memory blocks, can make the cache access time up to 100 times faster

Figure 2.1: Memory hierarchy example of a modern desktop. As we go further away from the processor, the memory becomes larger and slower. Source: The Author.



than the DRAM access time, as shown in Figure 2.1. Despite the higher performance, having large cache memories is infeasible for a set of reasons: SRAM technology is more expensive and occupies a larger area per bit since more transistors are necessary to store a bit; power and temperature constraints also need to be taken into account; furthermore, increasing the cache size can lead to an increase in the hit latency (FERDMAN et al., 2012; ESMAILI-DOKHT et al., 2018; NORI et al., 2018).Therefore, the size of cache memories in modern processors typically does not surpass 32 KB in levels closer to the processor and 2 MB per core in more distant levels (MORGAN, 2017; CUTRESS, 2017), in contrast with the several GB observed in DRAM memories. Unfortunately, the reduced size of cache memories prevents the application working set (a portion of data necessary to the application execution at a given moment) to fit inside the on-chip memory. Therefore, it results in accesses that miss the closer levels of the processor internal memory hierarchy, and the further a request needs to go into the memory hierarchy in order to be served, the higher is the latency and the power consumption.

In modern processors, a three level cache hierarchy is commonly used (MORGAN, 2017; CUTRESS, 2017). In this configuration, the first level data cache (L1) and the second level data cache (L2) are usually private to each processor core. These cache memories are closer to the processor, have less storage capacity, and provide more efficient data access. A third level of cache (L3, also know as Last Level Cache – LLC) is shared among the system cores. Its response time is frequently several times larger than

the private cache levels latency, but with the advantage of providing a larger storage capacity. When a processor issues a request for data to the memory, several situations can occur:

- Initially, the request is delivered to the L1 cache. This memory is relatively small (32 KB) and presents low latency (i.e., 4 processor cycles) (FOG, 2012; HENNESSY; PATTERSON, 2017). Simultaneously, the Translation Lookaside Buffer (TLB) is accessed to check whether the page is physically present in the main memory. The TLB returns the translation of the virtual address into a physical address (HENNESSY; PATTERSON, 2017). In the case that both information are found, the data is quickly delivered to the processor.
- If the address translation from virtual to physical address is not found in any TLB level, a page table walk must be performed; that is, the page table – stored in the main memory – needs to be accessed to check what is the virtual to physical address translation. Subsequently, an additional access to the main memory needs to be performed to obtain the data requested. These several main memory requests are very likely to profoundly harm the application performance.
- A second situation occurs when the address translation is found inside the TLB but the requested data is not present in the L1 data cache. In such scenario, it is not necessary to perform a page walk, but the data request now needs to be forwarded to the next level of cache memory, now using the physical address obtained through the TLB. The L2 is usually larger, typically around 256 KB, which increases the probability of finding the data but at the cost of a higher response time (FOG, 2012; HENNESSY; PATTERSON, 2017). The L2 cache repeats the procedure of searching for the requested data. If the data is found, it translates into a data request that misses L1 cache but hits L2 cache, taking 14 cycles to be delivered to the processor when added to the L1 latency (e.g., for the Skylake architecture (FOG, 2012)).
- If the data is not found in the scenario described above, it results in a much higher impact to the application performance due to the necessity of accessing the LLC. Although the LLC storage capacity is much bigger (around 2 MB per core, increasing even more the probability of finding the data), its access latency can be at the order of 80 processor cycles (FOG, 2012; HENNESSY; PATTERSON, 2017).
- Whenever the data request misses all three levels of cache (L1, L2, and LLC), the LLC then forwards the request to the off-chip main memory, which again harms the

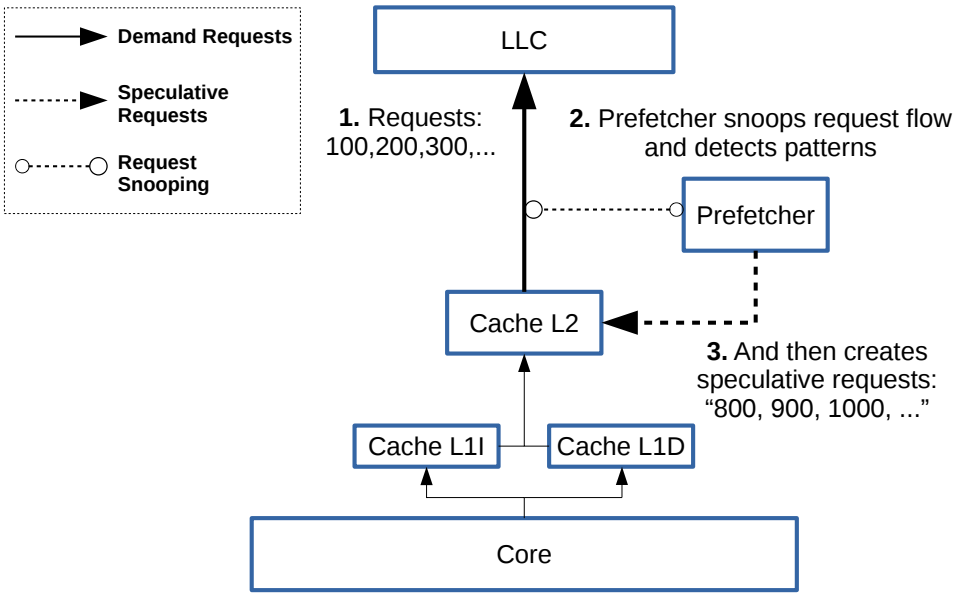
performance given the even greater access latency, a result of the aforementioned technological differences.

These technological constraints observed in the memory system are even more relevant when we consider the huge performance gap between memory and processor. In recent years, several improvements in the processor performance have been observed. An example is the increase in the number of cores, which requires memories with a higher data transfer bandwidth to handle the data requests issued by these several cores. We can also highlight the processor's ability of sending multiple instructions per cycle into the execution units (also known as multiple issue processors) (HENNESSY; PATTERSON, 2017), which may create the necessity of multiple demands for data in a single processor cycle. However, the memory technologies have not improved as much as processors, creating a performance gap referred in the literature as the memory wall (WULF; MCKEE, 1996).

Several problems can arise from such performance disparity. For instance, to correctly execute an application, two instructions that depend on one another must be executed in their original order. If the first instruction is a load and its required data is not quickly delivered by the memory system, the execution of both instructions might get stalled (HENNESSY; PATTERSON, 2017). To avoid such stalls, the load-to-use latency must be reduced, that is, the number of cycles from when the processor issues a load to the moment it can actually use the data should be as small as possible. Moreover, given the multiple issue nature of modern processors, a big number of memory requests can be issued in just a few cycles, possibly creating contention in some level of the memory hierarchy. Hence, finding the requested data in closer on-chip cache levels is preferable, otherwise, the memory hierarchy can become quite a bottleneck for the application performance (BAKHSHALIPOUR et al., 2019a).

In light of these several problems, the prefetcher hardware was added to mitigate the memory latency (BAER; CHEN, 1991). Prefetching, as a technique implemented in hardware, aims to predict which will be the next memory addresses to be requested by the processor. By monitoring the previous memory requests, a prefetcher is able to identify possible access patterns. Based on these patterns, it speculates which might be the next addresses to be requested and then performs the requests in advance, before the processor actually needs them. Thus, when the data is finally requested by the processor, it will already be in cache levels closer to it (HENNESSY; PATTERSON, 2017). The aforementioned main memory latency is therefore hidden by other processor instructions

Figure 2.2: Abstraction of the prefetcher behavior. Source: The Author.



previous to the effective load instruction that required the prefetched data.

With the data already at closer levels, (i) the critical load-to-use latency can be reduced (KANG; WONG, 2013; GUTTMAN et al., 2015), and (ii) an important performance metric is improved, the cache hit ratio. The hit ratio represents the portion of requests that are found in a given cache level without the need of going deeper into the memory hierarchy – and consequently requiring a higher execution time. These performance gains allowed prefetchers to become a prevalent mechanism in current architectures (MORGAN, 2017; CUTRESS, 2017; FOG, 2012). Examples of patterns identified by common prefetching mechanisms are stride (CHEN; BAER, 1995) and stream (LE et al., 2007).

Figure 2.2 shows an example of a L2 prefetcher detecting a stride access pattern. The L2 cache forwards requests to the LLC (shown in Figure 2.2 as the event **1**). The L2 prefetcher, in turn, intercepts these requests by snooping the cache interconnection (event **2**) and identifies the access pattern being generated. Based on the pattern identified, speculative memory requests are inserted into the L2 Miss Status Holding Register (MSHR) (**3**), a buffer that keeps track of miss events that still need to be handled. These speculative requests are made directly to the L2 cache in order to avoid a redundant cache fill if the speculated data block already resides in the cache. These accesses are seen as regular requests made to the L2 by the prefetcher, so the L2 does not actually need to forward the response to the L1. If the speculated address is not present in the L2 yet,

the next levels in the hierarchy will forward the response to whoever requested it, as in a regular access. Thus, when the processor needs the data requested by prefetch, it will already be at a closer cache level (in this case, the L2 cache).

Due to the layout of applications' data structures and the algorithm characteristics, several different types of memory access patterns are generated. Surveys regarding state-of-the-art prefetchers (MITTAL, 2016; BAKHSHALIPOUR et al., 2019b) classify these types of patterns accordingly with their complexity and the main pattern observed. For example, the *stride prefetcher* (CHEN; BAER, 1995) predicts the next addresses to be requested based on a regular stride information detected in the previous accesses. Considering that the last access sequence has followed the pattern A, $A + S$, $A + 2S$, $A + 3S$, then the next address to be requested might possibly be the $A + 3S + S$. When $S = 1$, we observe a *stream* pattern. However, many applications do not present such regular strided access patterns, being much more complex and hard to predict. *Irregular patterns* are often addressed by *correlation prefetching* techniques that aim to detect some spatial (SOMOGYI et al., 2006) or temporal (WENISCH et al., 2005) correlation among the accesses.

Aiming to cover as many different patterns as possible, a large number of prefetching techniques have been proposed over the years. *Strided prefetchers* (CHEN; BAER, 1995; JOUPPI, 1998; IACOBOVICI et al., 2004; ZHU; CHEN; SUN, 2010; KIM; ZHAO; VEIDENBAUM, 2014) take into account the constant distance between each address of a sequence of memory accesses. Such equally spaced addresses are usually found in applications with dense matrices and in applications that make use of pointer-based data structures where memory allocators arrange the data in a sequential manner. Due to the simplicity of the patterns, strided prefetchers demand small area and power overheads. However, as mentioned before, if accesses are irregular or with a more complex pattern, such prefetchers can be less efficient.

Spatial data prefetchers (Kumar; Wilkerson, 1998; NESBIT; SMITH, 2004a; CHEN et al., 2004; CANTIN; LIPASTI; SMITH, 2006; SOMOGYI et al., 2006; ISHII; INABA; HIRAKI, 2009; PUGSLEY et al., 2014; SHEVGOOR et al., 2015; KIM et al., 2016; MICHAUD, 2016; BAKHSHALIPOUR et al., 2019a) divide the memory into contiguous and fixed-size memory sections and rely on the spatial correlation perceived among these different memory pages. Many applications use complex data structures such as logs, buffers, and headers, where each field is usually of a fixed size and result in similar memory layouts. Therefore, different pages of the memory can be accessed similarly,

since the layout of the data in the cache translates into relative addresses with recurring patterns. Spatial prefetchers commonly store offsets (the difference between an address and the beginning of the memory page) or deltas (the distance between two consecutive addresses inside the same memory page) as metadata information, resulting in small area overhead. By using the patterns observed on previous memory pages to predict the accesses for a newly accessed page, spatial prefetchers perform very well on avoiding compulsory misses. On the other hand, they tend to perform poorly in applications with high usage of pointers, since dynamic objects can potentially be allocated in different memory pages where no pattern can be easily identified. It is worth noting that, although they may appear similar, spatial prefetchers are different from strided prefetcher. While strided prefetchers are only able to identify patterns with constant offsets, spatial prefetchers consider varying offsets between the address inside a given memory page.

Temporal data prefetchers (SOLIHIN; LEE; TORRELLAS, 2002; WENISCH et al., 2005; FERDMAN; FALSAFI, 2007; WENISCH et al., 2009; JAIN; LIN, 2013; BAKHSHALIPOUR; LOTFI-KAMRAN; SARBAZI-AZAD, 2018), in its turn, record information about the sequence of addresses that follow a miss into a history table, and use this history to predict future accesses. Upon a cache miss, the history table is searched for a matching entry, and the sequence of accesses is used to replay the sequence. Relating a sequence of addresses to a miss is very useful when considering dependent cache misses. Dependent cache misses are commonly found in pointer-chasing applications and refer to a cache miss that is dependent on the data of an earlier cache miss. However, since temporal data prefetchers rely on the repetition of the addresses, they are unable to avoid compulsory misses. Unlike spatial prefetchers, temporal prefetchers need to store the full correlation between accesses, requiring a larger area that usually does not fit in the on-chip region.

Several recent works introduce combinations of prefetching techniques. *Perceptron-based prefetchers* (BHATIA et al., 2019) propose a method to enhance the efficiency of usual prefetchers, such as the Signature Path spatial prefetcher (KIM et al., 2016). Moreover, many of the aforementioned prefetchers also use *heuristics* in their predictions (PUGSLEY et al., 2014; SHEVGOOR et al., 2015; KIM et al., 2016; MICHAUD, 2016). For example, (PUGSLEY et al., 2014) proposed the Sandbox Prefetch, a technique that defines at run-time the most efficient prefetching mechanism to a given program by making use of a Bloom filter. A Bloom filter is a probabilistic data structure employed to test whether an element is a member of a set in a memory-efficient way. Rather than

directly bringing the prefetched data into the cache, the address generated by a given prefetching candidate is added to the Bloom filter to check if this mechanism could have accurately prefetched the data. When the accuracy of the current candidate surpasses a given threshold, then real prefetches are performed.

2.2 Designing a Prefetcher: Constraints and Trade-offs

Despite the several benefits that data prefetching can bring to the system performance, the design of a prefetch mechanism must take into account a set of aspects and constraints. One common misconception is to believe that the more data blocks brought into the cache by the prefetcher, the better, presuming that this data would be covering more possible misses. However, that is not entirely true. The aggressiveness of a prefetcher is defined by the number of prefetches issued at once (what is known as *prefetch degree*) and by how far of the demand stream the prefetch speculates (known as *prefetch distance*) (EBRAHIMI et al., 2009). An overly aggressive prefetcher that is unable to correctly predict the future accesses of a given pattern might generate requests for data that will not be used. Since the cache size is fixed, these wrongly predicted data blocks might end up evicting from the cache data blocks with high probability of use in a near future. This circumstance is called *cache pollution* (MITTAL, 2016; BAKHSHALIPOUR et al., 2019b) and prior work proposed prefetching mechanisms that try to mitigate its occurrence (ZHUANG; HSIEN-HSIN, 2006; SRINATH et al., 2007; EBRAHIMI et al., 2009; ZHU; CHEN; SUN, 2010; WU et al., 2011).

Several works noticed that some applications performed better with more aggressive prefetchers, while others would benefit more from less aggressive mechanisms. Therefore, a set of related works proposed techniques that dynamically adapt the prefetchers accordingly to the system feedback. The work proposed by (NESBIT; DHODAPKAR; SMITH, 2004) divides the memory space address into zones and detect delta correlations between the miss addresses of different zones. On a program-phase fashion, the prefetcher degree and the size of the memory zones are dynamically adjusted.

An additional problem exacerbated by an overly aggressive prefetcher is the *cache thrashing* (JALEEL et al., 2010; MITTAL, 2016). The prefetcher-related cache thrashing occurs when the blocks predicted by the prefetcher will indeed be used, but still cause the eviction of other blocks that will also be used in a near future. This happens mainly because the working set of a given application does not fit in the cache limited capacity.

Moreover, the extra requests spawned by an overly aggressive prefetcher will also battle for memory bandwidth alongside demand requests (loads and stores), creating memory contention and significantly harming the system performance (SRINATH et al., 2007; EBRAHIMI et al., 2009; PUGSLEY et al., 2014). It is worth noting, however, that other reasons could be behind cache thrashing, such as the application access pattern and the cache replacement policy.

Regarding area and power constraints, the matter is much more complex than simply wasting potentially useful area on prefetching and consuming extra power. When an overly aggressive prefetcher generates cache pollution, the number of cache misses increases, requiring data requests to go further into the memory hierarchy. Consequently, the power consumption can increase dramatically (SRINATH et al., 2007; EBRAHIMI et al., 2009). Furthermore, several works pointed out that a given prefetcher design is only beneficial if its performance gains are able to outweigh its area overhead (LOTFI-KAMRAN et al., 2012; KAYNAK; GROT; FALSAFI, 2013; ESMAILI-DOKHT et al., 2018). Prior work also considered a metric called *performance density* (KAYNAK; GROT; FALSAFI, 2013; ESMAILI-DOKHT et al., 2018; BAKHSHALIPOUR et al., 2019a), defined as the *throughput per unit area*, quantifying how efficiently a prefetcher design uses the silicon area.

Another essential aspect that has several implications is the inter-core interference. With the increasing core count of multi-core systems, the prefetchers of each core will compete for shared resources. For instance, the conflicts among demand requests and prefetch requests from different cores might displace useful data from shared levels of cache. Moreover, prefetches from the entire system will require a portion of the available memory bandwidth, competing with the demand requests and again leading to memory contention (EBRAHIMI et al., 2009; PUGSLEY et al., 2014). As a straightforward consequence, both demand and prefetch requests can be delayed, affecting the delivery time of these requests in the whole system. Since the goal of a prefetch mechanism is to hide the memory latency, it is crucial to have the data in the cache soon enough to avoid a miss.

Considering the competition that may exist among the several cores, (EBRAHIMI et al., 2009) proposed a technique that adjusts the prefetcher aggressiveness in a coordinated way across the system. Their mechanism uses information of the inter-core interference to throttle the prefetcher in a hierarchical global-based way. First, the aggressiveness of each core is defined locally, attempting to optimize the local core performance. The global mechanism then considers information gathered from the entire system re-

garding inter-core interference on shared memories and on the system bandwidth, and can override the decision made by the local controller. The aforementioned Sandbox Prefetcher (PUGSLEY et al., 2014) defines at run-time the most applicable prefetching algorithm by adding to a Bloom filter the address to be prefetched. The mechanism can be applied to multi-core systems by rising the required accuracy before performing real prefetches.

The concept of *timeliness* of a prefetcher refers to how accurate it is in fetching the data at the right time (SRINATH et al., 2007; ZHU; CHEN; SUN, 2010). A prefetch is defined as *late* if the prefetched data is not yet in the cache when the processor requires it. Besides not contributing to the performance, it is also strengthening cache pollution and memory contention by uselessly requiring memory bandwidth. An *early prefetch*, by its turn, represents a prefetched block evicted from the cache by other blocks before being used by a demand instruction, mainly due to the limited cache size. Despite appearing merely useless, an early prefetch can also cause cache pollution and bandwidth overhead. With bandwidth contention affecting the timeliness and therefore resulting in more pollution, extra accesses to the off-chip memory might be necessary, again requiring extra power and increasing the load-to-use latency.

Prior work (CHOU, 2007) proposed an epoch-based correlation prefetcher to improve prefetching timeliness. Epochs consist in recurring periods of on-chip computation followed by off-chip accesses. Their proposal tries to eliminate the off-chip accesses of these epochs, therefore eliminating the epoch itself and improving the system performance. Using stream localization, (ZHU; CHEN; SUN, 2010) classify miss addresses into streams and maintain their timing information. Since the timing information of a particular access inside a stream is also predictable, the streams are later chained according to the time and this information is used when prefetching. A mechanism proposed by (SRINATH et al., 2007) considers the accuracy, timeliness and prefetch-related cache pollution to adjust the aggressiveness. Moreover, a run-time estimation of the cache pollution is performed and applied to decide where to place prefetched data into the LRU stack.

As we can notice, the several aspects regarding prefetching profoundly interact with each other. The prefetcher aggressiveness affects the cache pollution and the memory contention, that by its turn affects the prefetcher timeliness. If the prefetches are not performed in a timely manner, then we can again observe cache pollution, which straightforwardly resonates over the number of misses. Consequently, a bigger portion

of the available memory bandwidth is required and more off-chip accesses are necessary, intensifying power consumption. When considering multi-core systems, the complexity increases with the core count and the inter-core interactions. Therefore, a prefetching mechanism must consider several aspects in order to achieve performance gains.

2.3 Computer Architecture Simulators

In computer architecture research, analysis of physical implementation are infeasible due to the complexity and high cost for manufacture (SKADRON et al., 2003). Consequently, architecture simulators are considered the primary mechanism to implement and evaluate a new idea in this research field. In HPC systems, there are many other problems besides those inherent to the architecture. For instance, in order to communicate, several threads might require access to the same memory addresses in the application data structures. To maintain the program correctness, it is necessary to keep data coherence in the several cache levels of the different cores. Moreover, these memory interactions among different threads may also unpredictably change the data path through the memory hierarchy. Therefore, to develop and analyze new ideas that attenuate such problems that arise from parallelism, we require multi-core architecture simulators that support parallel workloads (AKRAM; SAWALHA, 2019).

As computational systems' complexity has increased over the years, accurately simulating entire systems has become a challenge. As a consequence, several different simulation techniques have been proposed to simulate a given characteristic appropriately. Each of these techniques distinctly balances simulation accuracy, low development efforts, and simulation speed. Although each aspect is relevant, a single simulator can hardly achieve all of them (DUBOIS; ANNAVARAM; STENSTRM, 2012), leaving the choice of which simulator to use for computer architecture researchers. The following sections classify state-of-the-art simulators in terms of their simulation detailing – and also as full-system or application-level simulation, their input, and as sequential or parallel simulators. The classifications are not mutually exclusive, meaning that a given simulator can apply one or more of the following techniques.

2.3.1 Simulation Detailing Classification

The depth of details simulated is a crucial element for classification. A highly detailed simulator requires significant development efforts and will probably demand several simulation hours. However, it is more likely to deliver a satisfying simulation accuracy than a simulator that implements fewer architecture aspects. The main detailing classes are functional, timing, and integrated functional/timing simulators. Functional simulators base their simulation model on the target architecture functionalities, not implementing microarchitectural specific aspects (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). By separating the functionalities and the logic behavior from the microarchitectural elements, functional simulators act like instruction set emulators. Given the less detailed nature of such simulators, simulation time is usually shorter. SimpleScalar (AUSTIN; LARSON; ERNST, 2002) is a toolset with various simulation models, one of which is *sim-safe*, an example of functional simulator. Simics (AARNO; ENGBLOM, 2014) also provides a functional simulation model and is able to simulate the program forwardly and backwardly. SimCore (KISE et al., 2004) is an Alpha processor functional simulator, and AtomicSimple (BINKERT et al., 2011) is a Gem5 functional model. Many simulators also apply binary instrumentation over the program binary, collecting information during the execution in a real hardware (EECKHOUT, 2010; AKRAM; SAWALHA, 2019). Based on the information obtained with binary instrumentation tools such as Pin tools (REDDI et al., 2004), the simulators then perform the functional simulation. Examples of simulators that employ this technique are CMP$im (JALEEL et al., 2008) and Sniper (CARLSON; HEIRMANT; EECKHOUT, 2011).

On the other hand, timing simulators perform microarchitectural simulation and are able to provide detailed statistics about the performance and the execution of a target system (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). Microarchitectural structures and their timing information are implemented, and several other components are configured by the user, e.g., the cache size and associativity, and the number of entries in the reorder buffer (ROB). Timing simulators can be classified as cycle-level, event-driven, and interval simulators. Cycle-level simulators drive the simulation based on each processor cycle, and therefore are usually slower and use more memory space if compared to functional simulators (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). SimpleScalar contains a cycle-level model called *sim-outorder*, which implements an out-of-order superscalar processor that sup-

ports speculation. MSim (SHARKEY; PONOMAREV; GHOSE, 2005) is a multi-thread timing simulator that implements major pipeline structures of the Alpha processor.

Event-driven simulators, on the other hand, are timing simulators that base their simulation on events instead of processor cycles (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). Instead of simulating cycle by cycle, they skip the time where no event occurs, jumping directly to the time when there is an event scheduled and allowing the simulation to be shorter. Examples of event-driven timing simulators are SESC (RENAU et al., 2005), which supports the MIPS instruction set architecture (ISA), and SimFlex (HARDAVELLAS et al., 2004), a cycle-level simulator that has some of its modules implemented in an event-driven fashion.

Finally, interval simulators are a type of timing simulators that divide the execution flow in the pipeline in intervals based on miss events (e.g., cache misses and branch mispredictions) (GENBRUGGE; EYERMAN; EECKHOUT, 2010; DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). The basis for interval analysis is the observation that, in the absence of miss events such as branch mispredictions and cache misses, a well-balanced superscalar out-of-order processor should smoothly stream instructions through its pipelines, buffers, and functional units (EYERMAN et al., 2009). Sniper (CARLSON; HEIRMANT; EECKHOUT, 2011) is a timing simulator that extends the original interval model by proposing the instruction-window centric (IW-centric) simulation model. The IW-centric model is a high-level core model that implements both interval modeling and a detailed simulation of the instruction window.

As an alternative, it is also possible to integrate functional and timing simulators in order to bring together the flexibility and accuracy of both models (AKRAM; SAWALHA, 2019). The two simulation types might or might not be coupled together in the same simulator. Simulators that couple the two models are known as *execute-in-execute* and are often much more complicated than the simulators which apply both techniques decoupled. However, this approach usually provides a more accurate simulation of timing-dependent instructions such as synchronization and I/O operations. A well-known simulator which couples both functional and timing models is gem5 (BINKERT et al., 2011). On the other hand, many simulators decouple the functional and the timing models to simplify the development. For instance, both SimFlex (HARDAVELLAS et al., 2004) and Gems (MARTIN et al., 2005) make use of Simics (AARNO; ENGBLOM, 2014) for the functional simulation, and Graphite (MILLER et al., 2010) and Sniper (CARLSON; HEIRMANT; EECKHOUT, 2011) both employ Pin (REDDI et al., 2004) for binary instrumentation.

2.3.1.1 Full-System or User-Level Simulation

Despite achieving higher accuracy, timing-simulators do not consider several aspects of the system as a whole and their impact on performance. For instance, the operating system might directly influence cache contents and thread migration, and the long latency access of hard disks can cause significant performance loss. However, these characteristics will not always be relevant or desired, since one might be interested in simulating an architecture without considering the operating system noise, for instance. Having that in mind, it is also possible to classify a simulator according to its target scope, that is, whether the entire system or only the processor component is being simulated. User-level simulators choose to simulate only the processor itself, ignoring other system components as I/O devices and co-processors (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). Whenever the application calls for an extra component, as an I/O device, only the architectural impact of the resource allocation is emulated. System calls are therefore simply bypassed to the host operational system. Examples of state-of-the-art user-level simulators are SimpleScalar (AUSTIN; LARSON; ERNST, 2002), Graphite (MILLER et al., 2010), Sniper (CARLSON; HEIRMANT; EECKHOUT, 2011), ZSim (SANCHEZ; KOZYRAKIS, 2013), and SiNUCA (ALVES et al., 2015).

Full-system simulators, on the other hand, are able to simulate the entire operating system and the application running on it as it would typically run on a real target machine. In the full-system simulation, all external components are simulated, such as I/O devices and network. Full-system state-of-the-art simulators can be exemplified by PTLsim (YOURST, 2007), Gem5 (BINKERT et al., 2011), and MARSS (PATEL; AFRAM; GHOSE, 2011).

The differences between the two techniques result in relevant characteristics that might influence the simulator choice. Designing user-level simulators is much simpler than designing full-system simulators, and implementing modifications into the user-level simulator code might as well be easier. Moreover, the simulation time on the user-level tends to be smaller. Furthermore, full-system simulation can introduce noise into the evaluation when it is not acceptable (ALVES et al., 2015). For instance, one might not be interested in evaluating how the operational system interferes with the values residing in the cache. Nevertheless, in the execution of parallel applications, the operating system plays an essential role in scheduling the threads, which might be relevant depending on the evaluation purpose.

2.3.2 Simulation Input Classification

State-of-the-art simulators apply two different methods to manage instructions to be simulated: trace-driven and execution-driven (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). In a trace-driven fashion, trace files contain prerecorded streams of the instructions executed by a given application. In order to record these instructions streams, the application must first be executed in an ISA-compatible processor. During the execution, the application is instrumented and the instructions are logged into the trace file by a trace generator (which is part of the simulator). In the case of executing in a multi-core system, the trace contains the instructions of each thread in an interleaved manner according to the order in which they occurred. The trace file is therefore provided as the input file for the simulator. Examples of trace-driven simulators are Shade (CMELIK; KEPPEL, 1994), which supports SPARC and MIPS systems and is also used for trace generation. Simplescalar (AUSTIN; LARSON; ERNST, 2002) also employs trace files for its simulation.

One drawback of this approach is the large size of trace files, which restricts the number of instructions in the trace file and may lead to a slow simulation time (SHARKEY; PONOMAREV; GHOSE, 2005). Usually, only a particular portion of the application is instrumented. Furthermore, simulators that employ the trace-driven approach are unable to accurately model branch misprediction events (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). Since the trace files do not contain the wrongly predicted-path, the simulator can only add a time penalty to the simulation, without the possible effects of the wrong-path prediction (e.g., a mispredicted branch that fetches the wrong instruction, requiring the pipeline to be flushed and consequently stalling the execution). An additional limitation of trace-driven simulators is the inability of modeling the non-deterministic behavior of parallel applications (AHN et al., 2013). Despite that, a few recent works propose trace-driven simulators of parallel architectures, such as Sinuca (ALVES et al., 2015), a FPGA-based simulation framework (ELRABAA et al., 2017), and SynchroTrace (SANGAIAH et al., 2018).

On the other hand, execution-driven simulators make direct use of binary files and executables instead of trace files (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019). The application is executed directly on the target system, enabling the simulation of wrongly-speculated paths. Some examples of execution-driven simulators are SimpleScalar (AUSTIN; LARSON; ERNST, 2002), Rsim (HUGHES et

al., 2002), Sniper (CARLSON; HEIRMANT; EECKHOUT, 2011), and ZSim (SANCHEZ; KOZYRAKIS, 2013). It is worth mentioning, however, that neither ZSim nor Sniper are able to simulate wrongly-speculated paths. Even though they are both execution-driven simulators, they are also Pin-based simulators, and the Pin tool does not record wrongly-speculated events.

2.3.3 Sequential or Parallel Simulation

With the continuous increase in core count inside the same monolithic chip – or integrated multi-processor –, it has become crucial to simulate multi-core and manycore systems. However, parallel architectures introduce many new aspects that must be taken into account when modeling simulators. For instance, a simulator must ensure the data coherence across the memory hierarchy of the several cores, as well as properly consider the contention that may arise in shared resources. Therefore, an additional aspect of classification is whether the simulation of parallel architectures is performed sequentially or in parallel is also a classification factor (DUBOIS; ANNAVARAM; STENSTRM, 2012; AKRAM; SAWALHA, 2019).

The sequential simulation relies on a single thread simulating the entire target architecture behavior, with the cores simulated in a round-robin manner. However, in a more fine-grained context, many structures require a specific simulation order since there may be dependencies among them. Therefore, the components inside a given core are simulated in a time unit of one cycle. First, we simulate one LLC cycle, and then we proceed to simulate one cycle of the L2, and so on. The same happens to each pipeline component until we are able to reproduce the pipeline effects through the core private caches. Once the simulation of one core cycle is finished, the same cycle is simulated in the next core, and so on. After simulating this one cycle for all cores of the target architecture, then the simulator must simulate the global events that may have been caused by each core individual simulation. Examples of such global events are inter-core communication, which requires forwarding the data through the interconnection network, and cache miss events on some core private cache levels, requiring the data to be sent to the shared LLC (DUBOIS; ANNAVARAM; STENSTRM, 2012). Some state-of-the-art simulators from this group are MINT (VEENSTRA; FOWLER, 1994), RSIM (HUGHES et al., 2002), GEMS (MARTIN et al., 2005), and SiNUCA (ALVES et al., 2015).

The parallel simulation, in its turn, makes use of several simulator threads to sim-

ulate the target architecture cores. By the nature of parallel programming, the design of parallel simulators is usually harder, but it allows a much faster simulation time. State-of-the-art simulators that exemplify this methodology are ZSim (SANCHEZ; KOZYRAKIS, 2013) and Sniper (CARLSON; HEIRMANT; EECKHOUT, 2011).

2.3.4 A Summary of Architecture Simulators

In the last Sections, we explored several distinct aspects that encompass the development of architecture simulators. Akram et al. (AKRAM; SAWALHA, 2019) categorize the current architecture simulators in terms of their supported hosts and targets, their pipeline model, whether they support multi-core systems, and according to their simulation technique. In Table 2.1, we present a fragment of this categorization, containing the simulators relevant to this study. The complete categorization can be found in (AKRAM; SAWALHA, 2019). For a matter of space, the categories are abbreviated as follows:

- **FUNC:** functional;
- **TIM:** timing;
- **EvDr:** event-driven;
- **Fsys:** full system;
- **UM:** user mode;
- **EDr:** execution-driven;
- **TD:** trace-driven;
- **MOD:** modular;
- **HMP:** heterogeneous multiprocessor.

Table 2.1: Architecture simulators summary table.

Simulators	**Supported Hosts (ISA/OS)**	**Supported Targets (ISA)**	**Pipeline Model**	**Multi-core Support**	**Category**
CMP$im	x86	x86	–	yes	Parallel UM cache
DRAMSim	x86/Linux	Input trace files	–	no	TD cycle-level DRAM
gem5	x86/Linux	x86, ARM, MIPS, Alpha, SPARC	in-order, out-of-order	yes	FSys/MOD/TIM cycle-level
GEMS	x86/Linux, AMD64-Linux, SPARCV9 (Solaris 8)	SPARC, x86	out-of-order	yes	FSys/TIM (decoupled functional and timing models)
Graphite	x86/Linux	x86	in-order, out-of-order memory completion	yes	parallel UM/TIM (decoupled)
MARSSx86	x86/Linux	x86-64	in-order, out-of-order	yes	FSys/TIM (decoupled functional and timing models)
McPAT	x86/Linux	Alpha, ARM, x86C SPARC	–	yes (HMP)	power, area, TIM
MINT	SGI, SPARC and DEC stations	MIPS	–	yes	UM/EDr
Multi2Sim	x86/Linux	MIPS32, x86, ARM, AMD, Evergreen, NVIDIA Fermi	out-of-order	yes (HMP)	UM/MOD/TIM
MSim	x86/Linux, Win2000, SPARC/Solaris	Alpha	in-order, out-of-order	yes (HMP)	UM/TIM (cycle-level)
PTLsim	x86/Linux	x86	out-of-order	yes	FSys/TIM (cycle-level)
SESC	Unix-based systems	MIPS	out-of-order	yes	UM/TIM/EvDr
SimCore	x86/Linux, Alpha, SPARC/Solaris	Alpha	–	yes	UM FUNC
SIMICS	Alpha, PPC, UltraSPARC, Linux/x86, Windows	x86 Linux, SPARC, Alpha Solaris, Windows, ARM, PPC	–	yes	FSys/FUNC
SimpleScalar	Linux/x86, Win200/x86, SPARC/Solaris	Alpha, Pisa, ARM, x86	out-of-order	no	UM/EDr/TIM
SiNUCA	x86-64/Linux	x86-64	out-of-order	yes (HMP)	TD/UM/TIM
Sniper	x86/Linux	x86, RISC-V	in-order, out-of-order	yes (HMP)	parallel UM/TIM
ZSim	x86-64/Linux	x86-64	in-order, out-of-order	yes	parallel UM/TIM

3 MOTIVATION

With the increasing necessity for higher computational power in the diverse areas of computing, multi-core systems are becoming more and more crucial. Applications deal with a growing amount of data, which makes the prefetcher even more relevant given its ability to hide the memory latency, directly impacting the system performance. Therefore, it is essential to examine how the prefetcher interacts with the different applications' access patterns in such complex systems with dozens of cores.

Mittal (MITTAL, 2016) provides a survey on the recent development of prefetching techniques up to 2016. In the survey, the authors describe the relevant prefetch metrics, such as accuracy, coverage, and timeliness (SRINATH et al., 2007), as well as the different types of approaches to improve prefetching, such as new pattern detection techniques (NESBIT; SMITH, 2004b), filtering prefetches (ZHUANG; HSIEN-HSIN, 2006), dropping prefetches (LEE et al., 2008), changing prefetches' priority on the memory controller (EBRAHIMI et al., 2009), and so on. When considering the elevated costs of chip development and manufacturing, it became fundamental to employ architecture simulators for such progress of prefetching research. However, given all these distinct techniques and the active development of new ones, one can hardly keep up with the design of prefetchers and additional techniques in the industry, which are not publicly disclosed to avoid competition and intellectual property breaches. Most simulator designers end up facing difficulties when modeling prefetchers, as they cannot detail all the techniques used in real hardware. Interestingly, many of these techniques are not tested in multi-thread applications where communication plays a major role in the memory hierarchy latency (JAIN; LIN, 2018; WU et al., 2019; BHATIA et al., 2019).

Previous works already pointed out several complications of working with computer architecture simulators (DESIKAN; BURGER; KECKLER, 2001; NOWATZKI et al., 2015). Due to the lack of explicit information that hardware companies employ to avoid competition and breaches of their intellectual property, it is difficult to obtain an accurate simulation that correctly presents all the characteristics of a processor and its architecture. Several works base their results on top of simulators that are almost treated as black boxes, with non-validated information and possible bugs and modeling errors. The detail level also plays an important role, with oversimplified models that simplify many key features, or an unnecessary detail level and possible overfitting, compromising the generalization to other hardware being simulated. Moreover, there is a tendency of

following the "overall trends" provided by the simulation, ignoring the impact of errors in specific details.

Since Desikan's work, several simulators have been created along with a validation effort to evaluate their accuracy. In Akram et al's research (AKRAM; SAWALHA, 2019), the authors evaluated and compared the gem5 (BINKERT et al., 2011), Multi2Sim (UBAL et al., 2012), MARSSx86 (PATEL; AFRAM; GHOSE, 2011), PTLsim (YOURST, 2007), Sniper (CARLSON et al., 2014), and ZSim (SANCHEZ; KOZYRAKIS, 2013) simulators. After thorough characterization of the simulators, the authors performed experiments with single-core and multi-programmed workloads on each simulator. The results of each simulator were compared with Intel Haswell architecture (HAMMARLUND et al., 2014). The authors then highlighted the simulators' error sources, their sensitivity to different architectural parameters, and their relative error. Thus, they concluded that the lack of validation of a simulator for the target architecture, no matter how popular it is (e.g., gem5, which was only validated for ARM (GUTIERREZ et al., 2014)), leads to low accuracy and may render experiments invalid due to erroneous conclusions. Therefore, despite the validation against a specific architecture, one can not generalize the validation results or simply change configuration parameters expecting a similar performance and behavior for a distinct target (NOWATZKI et al., 2015).

A more recent work (WALKER et al., 2018) automates the process of finding the source of simulator error. The main objective of the work is to obtain more accurate energy consumption models for gem5, but to do so they required a more accurate processor model. Using gem5 and the processor configurations provided by Gutierrez et al.'s gem5 validation (GUTIERREZ et al., 2014), Walker et al. created GemStone, a framework to find the sources of simulation error based on empirical hardware Performance Monitor Counters (PMCs) models. GemStone selects the events in gem5, correlating them with the PMC events, and at last, performs one regression analysis to approximate the relationship between hardware PMCs and gem5 error, and another to approximate the relationship between gem5 events and gem5 error.

Given these substantial disparities between the real hardware and architecture simulators, many works might have their results questioned. It becomes challenging to attest how accurate and representative a given prefetching mechanism is if it is built upon a non-validated simulator. Even if the simulator is validated, this process is frequently based on top of a specific architecture, again confusing the assessment of the study correctness. Moreover, simulator designers often need to choose which key features will receive a big-

ger focus and detail level, which might produce oversimplified models that culminate in several inaccuracies and wrong conclusions.

To further complicate the matter, most state-of-the-art prefetchers and simulators are built upon single-core systems, which are less and less prominent in everyday scenarios. As mentioned in Chapter 2.2, it is challenging to understand the entire behavior and effects of prefetching algorithms over the system performance. One needs to take into account several aspects, many of those who are intensified in multi-core systems. The mechanism aggressiveness level (which can exacerbate cache pollution and cache thrashing), area and power constraints, the inter-core interference highly present in multi-core systems (increasing shared resource competition and causing system contention), and several other conditions deeply interact with each other. Therefore, given these various challenges, many simulators do not validate their prefetcher implementations or validate it only against sequential applications (CARLSON; HEIRMANT; EECKHOUT, 2011), while many others not even model the prefetcher (SANCHEZ; KOZYRAKIS, 2013). Together with the necessary design simplifications, the lack of validation can result in misleading conclusions in several works related to the memory system.

Through a careful experimental campaign, we ambition to fill some gaps in understanding how these aforementioned aspects relate to each other in such complex systems with multi-level memory hierarchy and with several processing cores. We investigate the performance of the different prefetchers available in the real hardware and analyze the system behavior when increasing the application parallelism. The behavior of multi-core architecture simulators and their prefetching techniques is also addressed in this work, comparing to the performance and measurements from a real hardware environment with multi-thread high-performance applications. At the end of this work, we hope to bring further knowledge on the characteristics of the complex multi-core systems prefetching and their memory hierarchy.

4 METHODOLOGY AND EXPERIMENTAL ENVIRONMENT

In this Chapter, we discuss the entire methodology campaign adopted in this work. We introduce details regarding the simulators employed, such as their implementation choices and validation. Moreover, we describe the experimental environment, covering the available prefetching algorithms provided by both the real machine hardware and Sniper, and the different prefetcher combinations executed and simulated, following with the methodology applied in the experiments and the benchmark used.

4.1 Simulators

As already discussed, computer architecture simulators play a crucial role in developing and analyzing new ideas in computer architecture research. As presented in Section 2.3, it is possible to classify state-of-the-art simulators according to many different parameters since each of them approaches a given set of architectural aspects and proposes distinct implementation techniques. However, depending on the detailing level and on implementation choices, many simulators may present high error rates and inaccuracies, compromising the simulation validity. When we consider multi-core systems, the question is further complicated since many other aspects arise with parallelism and are sometimes harder to be implemented in simulators. Therefore, this work proposes an evaluation of the prefetcher simulation accuracy of two parallel architecture simulators, ZSim (SANCHEZ; KOZYRAKIS, 2013) and Sniper (CARLSON et al., 2014). Details about the simulators are depicted in the following sections.

4.1.1 ZSim

ZSim was selected for our study due to its speed and accuracy when simulating x86 architectures, characteristics presented in its validation study (SANCHEZ, 2016). It is an instruction-driven simulator that uses Dynamic Binary Translation (DBT) to perform the instruction execution and dynamic instrumentation. The simulation uses a two-phase method called Bound and Weave. In the Bound phase, a few thousand cycles are simulated, ignoring the contention and applying a minimal latency for all memory accesses. In this phase, a trace of all memory accesses is recorded, including which caches lines

were accessed, evicted, invalidated, and so on. In the Weave phase, a parallel simulation is performed, oriented to the events of the recorded interval to determine the actual latency of each memory request in each component. Once the interactions between memory accesses have already been identified in the first phase, this timing simulation of these accesses can be done efficiently, maintaining high precision.

The authors observed that, in an interval of a few thousand cycles, most of the concurrent accesses between different cores happen to unrelated cache lines. Therefore, simulating these unrelated accesses first out-of-order and ignoring contention and, later, simulating them in order respecting time constraints, is equivalent to simulating them entirely in order. However, when accesses are related, i.e., access the same cache line, it is necessary to maintain the coherence of the different copies of the data in the different cores. An example of this is when a core demands exclusive access over a shared cache line to write into it, causing the cache coherence protocol to invalidate the other copies of the line in the cache hierarchy system.

The set of requests and messages necessary to invalidate the other copies of the line to obtain it as exclusive is known as Request for Ownership - RFO. However, for being considered rare, the order of these accesses to the same line is not modeled by ZSim, which can change the path of this data in the cache hierarchy. Changing the path of the data through the cache can impact the number of cycles, misses, and coherence messages observed in the simulation. In addition, the generation and the paths of prefetch requests can also change, preventing the modeling of prefetcher in this simulation model.

In its validation, ZSim uses the benchmark PARSEC (BIENIA et al., 2008) and shows only that the speed-up is close to that obtained with real executions by varying the number of threads. Therefore, since the prefetcher is not simulated, we seek to understand the impact of its absence and evaluate the accuracy of the Bound and Weave simulation method used to simulate multiple structures and threads in parallel.

4.1.2 Sniper

Sniper (CARLSON et al., 2014) is a parallel architecture simulator that extends the original interval simulation model (GENBRUGGE; EYERMAN; EECKHOUT, 2010). The authors' proposal is the instruction-window centric (IW-centric) simulation model, an approach that puts together the interval modeling with a more detailed simulation model of the instruction window. Through a detailed analysis of the micro-op dependency at

issue stage, their extension improves the simulation accuracy with a low increase in simulation time. Moreover, an improved memory access dependency analysis is performed, differentiating instructions that depend on long-latency loads and those which do not.

The basis for the interval simulation is the concept that, if no miss events occur (such as branch mispredictions and cache misses), then the execution should continue smoothly throughout the processor pipeline, functional units, and buffers (EYERMAN et al., 2009). However, this can only be observed under unrealistic conditions. Therefore, the effect of such miss events is the division of the execution into intervals, which serve as the fundamental entity for analysis and modeling for the interval simulation. The original interval modeling is itself a simulation model that allows the simulation of prefetchers in Sniper.

The interval simulation works with two structures named new window and old window (CARLSON et al., 2014). Each window admits as many micro-ops entries as exist in the reorder buffer of an out-of-order processor. While the new window represents the upcoming micro-ops and is completely filled the entire time, the old window contains a list of the most recently dispatched micro-ops. Through the micro-ops found at the new window, it is possible to identify memory level parallelism (MLP). The instruction level parallelism (ILP), on the other side, is calculated by analyzing the critical path of the old window micro-ops list.

In its validation, Sniper uses the benchmark SPLASH-2 (WOO et al., 1995) and also shows that the average speed-up error is small when varying the number of threads. Although Sniper provides an implementation of the prefetcher behavior, it has only been recently validated for the Cortex-A53 and Cortex-A72 ARM cores (ADILEH et al., 2019). The prefetcher model is responsible for over half of the simulation error in the *povray* and *x264* benchmarks, as some implementation details of the real hardware are unknown. However, the validation used only sequential benchmarks. In contrast, this study contributes to a better understanding of the accuracy of the simulated prefetch model and how it behaves in simulations of parallel architectures.

4.2 Experimental Setup

To collect information about the application execution in the real machine, we make use of the PAPI (TERPSTRA et al., 2010) tool. PAPI allows the obtaining of hardware counters values, a set of registers that provide information about CPU events such

as the number of instructions and the number of cycles. In each real execution, only one hardware counter is evaluated, thus avoiding aggregations or approximations that the tool performs when calculating several metrics in the same execution. The performance and the real executions statistics were evaluated with 10 executions of each metric, for each benchmark. The execution environment was composed of an Intel Xeon Silver 4116 CPU with 2.1 GHz of frequency, the Skylake microarchitecture (DOWECK et al., 2017). The simulation environment is an approximation of the real hardware, respecting the simulators limitations. Each simulation was executed only once, and all statistics were extracted from the same deterministic simulation. All the executions in the real machine were performed with the Intel Turbo Boost (ROTEM et al., 2012) technology disabled.

An important observation regarding the Skylake architecture is the change from an inclusive L3 cache to a non-inclusive one. The Skylake predecessor architecture was Intel Broadwell, which presents an inclusive L3 cache, meaning that all data brought into the L2 cache is also brought into the LLC. However, in non-inclusive L3 caches like the one used by Skylake, the data found at the L2 cache may or may not be found in the LLC, and there is no guarantee regarding how it will behave. Which data is brought to which level depends on the application access pattern, code and data sizes and their layout in the memory, and also on the inter-thread communication and sharing behavior.

Table 4.1 presents the configuration of the real machine and the processor simulated by ZSim and Sniper. The Sniper out-of-order core model was based on the Nehalem architecture (CARLSON; HEIRMANT; EECKHOUT, 2011), while ZSim based its out-of-order core model implementation on the Westmere architecture (SANCHEZ; KOZYRAKIS, 2013), a process shrink of Nehalem. Therefore, both simulators present a 16 stages pipeline, while the real machine architecture may present between 14 and 19

Table 4.1: Real machine, ZSim and Sniper configurations.

	Real	**ZSim**	**Sniper**
Processor Frequency	2.1 Ghz	2.1 Ghz	2.1 Ghz
Number of Cores	12	12	12
Pipeline Stages	18	16	16
Cache Line Size	64 B	64 B	64 B
L1 Data Cache	8-way 32 KB	8-way 32 KB	8-way 32 KB
Latency	4	4	4
L1 Instruction Cache	8-way 32 KB	8-way 32 KB	8-way 32 KB
Latency	4	4	4
Unified L2 Cache	16-way 1 MB	16-way 1 MB	16-way 1 MB
Latency	ca. 14	14	14
Last Level Cache L3	Non-inclusive 11-way 16.5 MB	Inclusive 11-way 16.5 MB	Inclusive 11-way 16.5 MB
Latency	ca. 60-80	77	77
Prefetchers	L1 IP Stride Adjacent Line prefetcher L2 DCU Stream prefetcher	No Prefetcher	Simple (Stride L1 Prefetcher) Global History Buffer (L2 Prefetcher)

stages (FOG, 2012). Other parameters such as cache associativity and cache access latency are easy to configure in the simulators and can be found in (FOG, 2012; DOWECK et al., 2017).

In Table 4.2, we present the description of the prefetcher algorithms considered in this work. The algorithms found in the L1 cache hardware are the Data Cache Unit (DCU) Prefetcher (INTEL, 2019) and the DCU IP Prefetcher (INTEL, 2019). The DCU Prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line. The DCU IP Prefetcher keeps track of individual load instructions (based on their instruction pointer value). If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride.

The L2 Hardware Prefetcher (INTEL, 2019) and the L2 Adjacent Cache Prefetcher (INTEL, 2019) are the prefetcher algorithms found in the real machine L2 cache. The L2 Hardware Prefetcher monitors read requests from the L1 cache for ascending and descending sequences of addresses. Monitored read requests include L1 data cache requests initiated by load and store operations and also by the L1 prefetchers, and L1 instruction cache requests for code fetch. When a forward or backward stream of requests is detected, the anticipated cache lines are prefetched. This prefetcher may issue two prefetch requests on every L2 lookup and run up to 20 lines ahead of the load request. The L2 Adjacent Cache Prefetcher fetches two 64-byte cache lines into a 128-byte sector instead of only one, regardless of whether the additional cache line has been requested or not.

The prefetcher algorithms in Sniper are the Simple and the Global History Buffer (GHB) (NESBIT; SMITH, 2004b). Based on an analysis of the Simple prefetcher code, we observed that it is similar to a strided prefetcher algorithm. Thus, we use the Simple prefetcher as the L1 cache prefetcher in our experiments. The GHB prefetcher is an n-entry FIFO table that holds the n most recent L2 misses addresses. Each GHB entry stores a global miss address and a link pointer that is used to chain the GHB entries into address lists. Each address list is a time-ordered sequence of addresses issued by the

Table 4.2: Prefetcher algorithms.

Prefetchers	Description	L1	L2	Real	Sniper
DCU Prefetcher	Streaming prefetcher, fetches the next cache line into L1D Cache	**X**		**X**	
DCU IP Prefetcher	Strided prefetcher of next L1D line based upon sequential load history	**X**		**X**	
L2 Hardware Prefetcher	Mid Level Cache (L2) streamer prefetcher		**X**	**X**	
L2 Adjacent Cache Prefetcher	Prefetching of adjacent cache lines into L2 Cache		**X**	**X**	
Simple	Strided prefetcher of L1D line	**X**			**X**
Global History Buffer	L2 Prefetcher based on global miss addresses		**X**		**X**

same instruction pointer. Therefore, based on the information of the address lists, it is possible to implement a correlation based prefetcher (CHARNEY; REEVES, 1995) and a stride prefetcher (NESBIT; SMITH, 2004b). The L2 prefetcher found in Sniper is a GHB correlation based prefetcher, and in our experiments it is used as the L2 cache prefetcher.

The hardware prefetchers present in Skylake can be enabled and disabled by changing the values of Model Specific Registers (MSR) through the instructions RDMSR (read MSR) and WRMSR (write MSR). MSRs are a set of control registers used for debugging, program execution tracing, computer performance monitoring, and toggling certain CPU features (INTEL, 2021). Therefore, based on the aforementioned prefetchers, we conducted experiments on the real machine considering the following configurations:

- All Skylake L1 cache prefetchers;
- All Skylake L2 cache prefetchers;
- All Skylake prefetchers from both L1 and L2 cache;
- No prefetcher.

Based on the Sniper prefetchers, the following systems were simulated:

- Only the L2 data prefetcher (GHB);
- Both L1 (Simple) and L2 (GHB) prefetchers;
- No prefetcher.

Since ZSim does not model the prefetcher behavior, there are no variations of the simulated system. With ZSim we only perform simulations with no prefetcher, which we refer to as ZSim.

4.3 NAS Parallel Benchmarks

For this study, we used a known HPC benchmark in the literature called Numerical Aerodynamic Simulation Parallel Benchmark (NPB) (JIN et al., 1999). This set of applications comprises ten applications, each of which encapsulates a certain type of computation that is often processed by HPC applications, e.g. computational fluid dynamics (CFD), adaptive meshes, parallel I/O, and computational grids. We used nine of the ten NPB applications, namely: CG (Conjugate Gradient), EP (Embarrassingly Parallel), FT (Fourier Transform), IS (Integer Sort), MG (Multi-Grid), UA (Unstructured Adaptive mesh), BT (Block Tri-diagonal solver), LU (Lower-Upper Gauss-Seidel solver), and SP

(Scalar Penta-diagonal solver). The Data Cube (DC) application was discarded because DC mainly stresses I/O operations, which are not modeled by the architecture simulators. All NPB applications are optimized to perform homogeneous memory accesses, with a small or nonexistent load unbalance.

The benchmark set presents different input classes that offer different input sizes and complexities. The available input classes are: S, W, A, B, C, D, E, and F. The class S was designed for quick testing purposes; class W was originally designed for standard testing considering a 90's workstations, and nowadays is used for quick test purposes as well; classes A, B, and C are standard test problems, whose size increases four times from one class to another; and classes D, E, and F, for large-size problems, whose size increases sixteen times from one to another. Most of the experiments in this work were made using class A since it is large enough to make assessments over the memory system of the Skylake architecture and small enough to avoid exceeding the feasible simulation time.

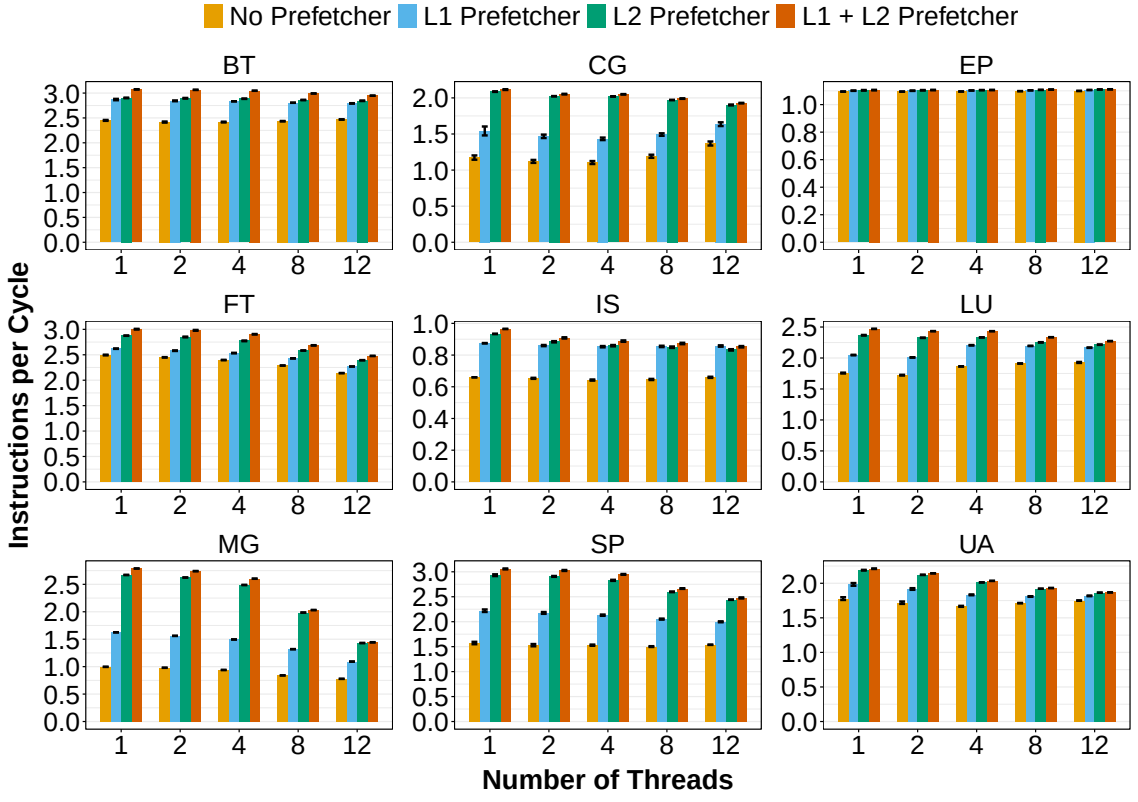
5 INVESTIGATING CURRENT ARCHITECTURE PREFETCHERS

In this Chapter, we present the main results obtained through the aforementioned experimental setup, taking into account the real execution of the NPB benchmark. More specifically, here we shed light to the following question: *How do the different prefetchers affect the execution of NPB over a real machine with a varying thread level parallelism?*

Figure 5.1 shows the instructions per cycle (IPC) for each NPB application, using the input class A, taking into account different numbers of threads and prefetchers, and with hardware prefetcher disabled as well. The IPC can be understood as a general performance metric of the prefetchers when executing the applications, since an efficient prefetcher will enable more processed instructions per cycle by fetching the right data at the right time. The error bars represent the variability observed among both the cores of a certain execution and the 10 distinct repetitions of the executions.

As expected, we can observe that any prefetcher increases the execution performance of NPB, with the exception of the EP application. Since EP is known to have a small memory footprint (JIN et al., 1999), processor stalls due to memory access latency

Figure 5.1: IPC results for the real execution of the NPB applications with input class A. Standard deviation lower than 5%.



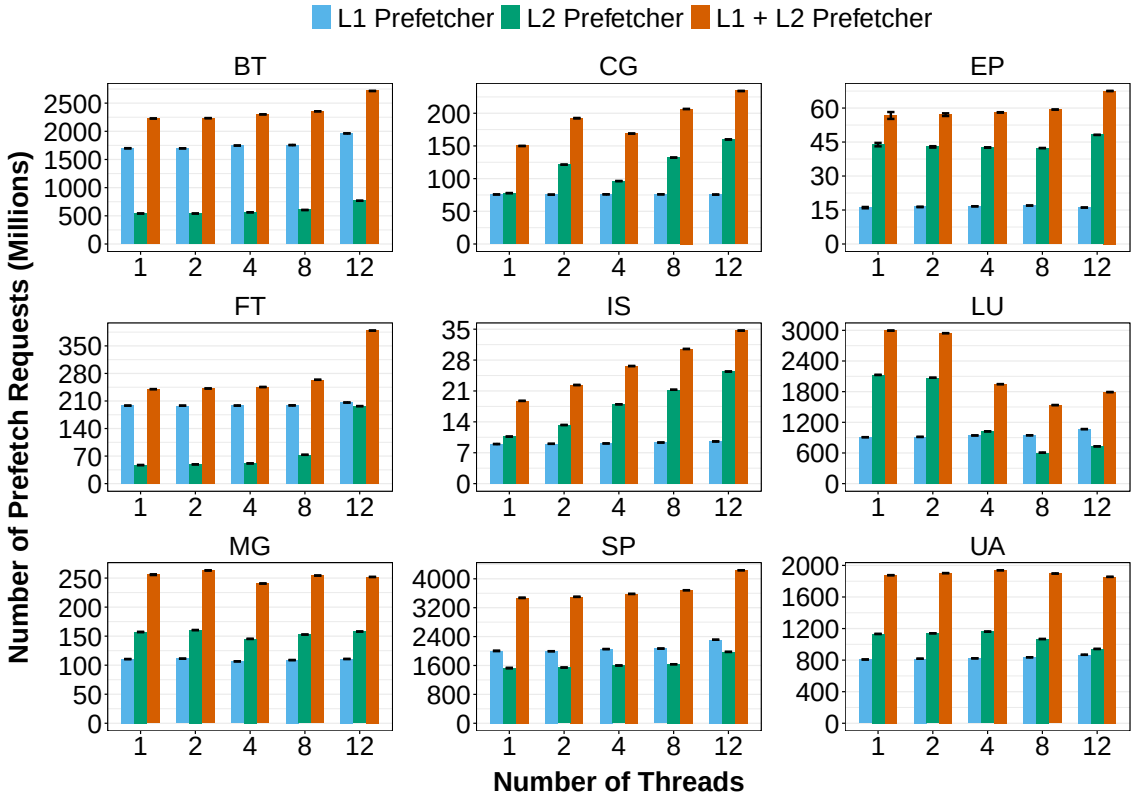
rarely occur during its execution. Therefore, there is little difference in the IPC given different prefetcher configurations for EP. For the other applications, memory prefetching significantly contributes to a better IPC, specially for executions up to four threads.

However, the increase in performance from a standalone L2 prefetcher to the combination of L1+L2 prefetchers is not as large, with an overall average IPC improvement of 2.9%, when compared to the 31.9% of performance increase from no prefetcher to the standalone L2 prefetcher. This can be explained by the ability of the L2 prefetcher to detect more relevant memory access streams that are dependent on the LLC long-latency response, while the L1 prefetcher detects access to data that might be found at the on-core L2 cache level. Since the difference in latency from the L1 to the L2 is small, and the latency from the L2 to the LLC is much larger, this means the main performance gains would be obtained by the L2 prefetcher and the associated access streams it detects. For concrete numbers, the L2 cache access latency is of 14 cycles, only 10 cycles higher than the L1 cache, while the LLC latency in Skylake is measured to be approximately between 60 and 80 processor cycles, presenting a much more substantial overhead and a higher probability of stalling the processor execution (ALVES et al., 2015). Furthermore, in Figure 5.2 we present the sum of all prefetch requests performed by the active prefetchers of each active core in the real execution, also for the input class A. We can observe that, for some applications (e.g. BT, FT, SP, and some executions of LU), the number of prefetch requests performed by the L1 prefetcher is larger than the number of requests performed by the L2 prefetcher, and, in some cases, it almost reaches the same amount of requests performed by the two prefetchers combined (the L1+L2 executions). Despite the L1 prefetcher issuing more prefetches than the L2 prefetcher in several cases, the fewer L2 prefetches are the ones who deliver the most crucial performance gains, as seen in Figure 5.1.

Another complementary explanation for these small performance gains observed for the L1 prefetcher is that its requests may be detrimental to performance, since they may compete with demand data for the rather limited cache space on the L1 cache level, as pointed out on Section 2.2. Moreover, these prefetch requests may also occupy too many entries on the line fill buffers present in the L1 hardware, competing again for shared and limited resources with the critical demand accesses, which are more relevant for the immediate processor execution than the prefetched lines.

The L1+L2 prefetcher combination is often set as the default setting in the machine configurations. One of its main appeal is that it is the setting that provides the

Figure 5.2: Number of prefetch requests of the real machine executions with the input class A. Standard deviation lower than 5%.



best performance. However, one drawback of this approach lies in the interpretability of what is being performed by the algorithms. The lack of understanding over the full prefetcher hierarchy behavior hinders the accurate implementation of prefetching models in architecture simulators, which will be demonstrated in detail in Chapter 6. For instance, the interactions between the prefetchers of each cache level (e.g., whether the L2 prefetcher considers L1 prefetches as part of the application access stream or as prefetch requests) can cause major changes in the behavior of the memory hierarchy. In contrast, the L2 prefetcher in isolation provides a simpler, more transparent understanding of the prefetches performed by the algorithm, facilitating reverse engineering and reproduction or simulation of the prefetcher, almost reaching the best observed performance (L1+L2). At the light of these results, it may be advisable to set the L2 prefetcher on its own, as opposed to the L1+L2 setting, since the lack of transparency in the details of the L1 and L2 prefetcher, the L1 line fill buffer contention, and the additional energy consumption of the L1 prefetcher may not outweigh its increase in performance.

Another interesting trend observed in Figure 5.1 is the decrease in the IPC for an increasing number of threads in the majority of applications and prefetchers, with the EP application being an exception. Apart from EP – which, for the same reasons explained

above, the IPC is stable for any number of threads on all cores – the NPB applications use memory accesses and inter-thread communications, that naturally increase in function of the total number of threads. With a large number of threads, these memory accesses and inter-thread communications generate contention that become a larger constraint in the IPC, and the performance provided by prefetchers becomes small or negligible. This effect can be clearly seen in the IPC graph for the MG application: MG is known to use memory-intensive operations and to be communication-intensive, largely benefiting from memory prefetchers. However, as the number of threads increases, the contention becomes a more considerable constraint in the IPC results, significantly harming the performance.

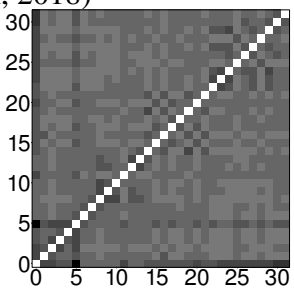
In this regard, we hypothesize that, when the number of concurrent threads executed in a processor is high, the performance benefit from memory prefetchers for applications that use intense memory and inter-thread communications is negligible. These characteristics would need to be alleviated on the applications for memory prefetchers to be effective. However, an interesting exception to this observation are the executions of the CG application with the L1 prefetcher and without prefetcher (Figure 5.1), where the IPC increases in function of the number of threads. We explain the CG case with more details in Section 5.1.

5.1 The CG Case

As previously explained, all NPB applications, except CG, suffer from a performance decrease as a result of the increasing memory contention that harms the prefetcher efficiency in highly parallel applications. However, the CG executions with the L1 prefetcher and without prefetcher present a contrasting behavior, with the IPC increasing in function of the number of threads, as demonstrated in Figure 5.1. This section aims to detail the CG executions to better understand the CG results, showing how the application behavior contributes to the different observations.

First, in Figure 5.3 we can observe the communication pattern found in CG. In this figure, the darker a given cell is, the more shared memory accesses occurred between the threads represented by row and column. The diagonal is filled with white as a thread does not share data with itself. Thus, we can see that all threads of CG irregularly communicate with all threads. These irregularly distributed memory accesses enable one thread to request a memory address that had already been requested by another thread or even by the prefetcher of another thread. Since the chosen processor model has a non-inclusive

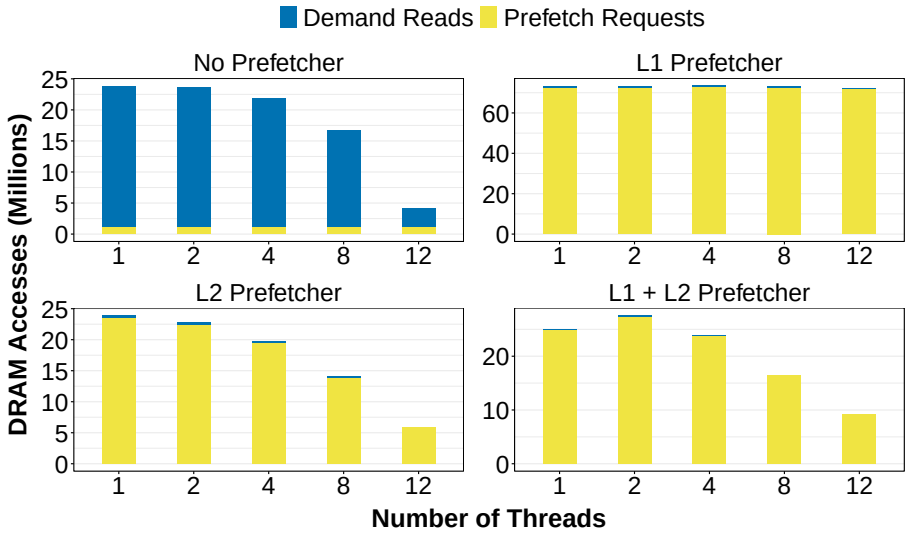
Figure 5.3: CG communication pattern for 32 threads. Figure obtained from: Cruz *et al.* (CRUZ; DIENER; NAVAUX, 2018)



L3 cache (as mentioned in Section 4.2), the data found in the L2 cache is not necessarily duplicated at the L3 cache as would happen in an inclusive cache hierarchy, resulting in more available cache space. Therefore, as we increase the number of threads, the amount of cache space available for the application is increased by using each core private cache levels (which include an 1 MB L2 cache per core). Consequently, a larger part of the working set of the application fits inside the processor caches, reducing the number of LLC misses, and thus reducing the number of long-latency DRAM accesses as well.

This behavior is shown in Figure 5.4, where we depict the number of DRAM accesses as we increase the number of threads. The instruction prefetcher has not been disabled for any of these executions, which explains the presence of prefetch requests when the hardware data prefetchers are disabled. As we use more threads, fewer DRAM accesses per core need to be performed regardless of the prefetching mechanism, precisely because the application has the necessary data present in the cache. Moreover, a more prominent decrease is observed on the demand read DRAM accesses for the executions without prefetcher. These avoided DRAM accesses play an essential role in the performance improvements observed in these executions. A prefetcher is accurate if it can hide the main memory latency by effectively pulling data from the main memory to cache memory before a demand access requests the data. An accurate prefetcher should also generate a number of prefetches similar to the number of demand requests generated by the processor without prefetchers, i.e., the actual number of load/store instructions that would require data. Otherwise, it is generating unnecessary main memory accesses to bring data that will not be used. This does not necessarily reduce the IPC, but it can increase main memory contention, energy consumption, and cache pollution. Within this context, Figure 5.4 shows that the L1 prefetcher is quite inaccurate, generating many more speculative accesses to the DRAM in comparison with the execution without prefetcher. In stark contrast, the L2 prefetcher on its own is a lot more accurate, as the total number of DRAM accesses for each number of threads is close to the case of no prefetching.

Figure 5.4: CG main memory accesses per core, originated by demand reads and prefetches requests, for input class A. Standard deviation lower than 5%.

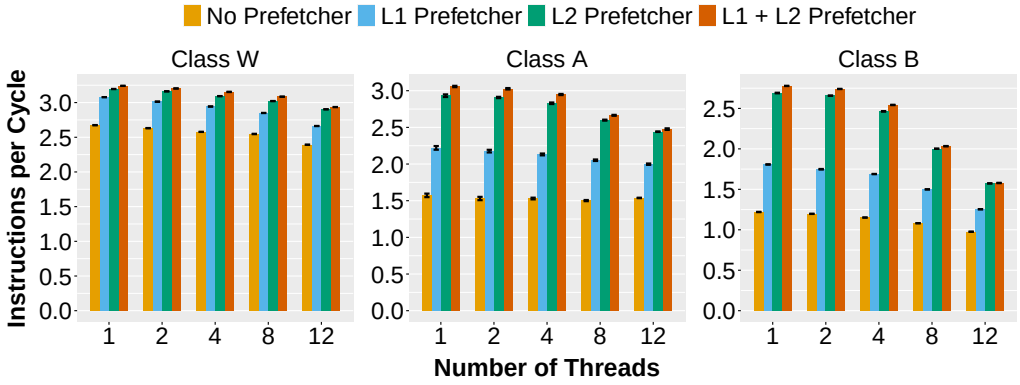


In Figure 5.1, we also notice a contrast in the performance of configurations with the L2 prefetcher, and those without. Configurations with the L2 prefetcher do not have an IPC increase with a higher number of cores, as the L2 prefetcher is correctly speculating and bringing the data to the caches even with a low core count and low amount of cache. On one hand, the IPC tendency of configurations with an L2 prefetcher is to decrease, as the prefetcher effectively hides the latency of DRAM accesses, but increasing the number of cores generates contention in the DRAM access. On the other hand, configurations without the L2 prefetcher cannot hide this latency, and their IPC increase as the application is able to retain larger portions of the working set in higher quantities of private non-inclusive L2 caches.

5.2 Effects of Different Input Classes

Up to this point, we analyze the relationship between the prefetcher and the parallelism increase with fixed input size. However, the memory requirements of different HPC applications may vary substantially, requiring different execution parameters. For instance, when considering small input sizes, an execution with several threads may highlight communication effects, which will play a decisive role in the application performance. Simultaneously, the prefetcher effectiveness may also be disturbed if we increase the parallelism in communication-intensive applications. In this section, we aim to better understand how the prefetcher influences performance when we vary the application memory requirements.

Figure 5.5: IPC results for the real execution of the NPB SP application, using input classes W, A, and B. Standard deviation lower than 5%.



In Figure 5.5, we present the IPC results for the SP application with different thread counts, different hardware prefetcher configurations, and variable input sizes, using NPB classes W, A, and B. We present the results only for the SP application since the same behavior is observed for all other NPB applications. The first point to be noted is that as we increase the problem size, the prefetcher influence on the application performance increases as well. If no prefetcher is enabled, class W suffers from a modest performance loss, while classes A and B are able to improve their performance by up to two times with all prefetchers activated. Since class W has smaller memory requirements, any prefetcher that can predict a simple access pattern may already be able to fetch a significant part of the application working set in advance. Therefore, the performance decrease observed as we increase parallelism in class W is caused by the thread communication overhead observed over small input sizes, which can not be entirely solved by the prefetcher.

When we compare the performance reduction observed with the increase of parallelism for classes A and B, we identify that a bigger problem size suffers from a more dramatic loss. As we increase the parallelism of the execution with the class A, the execution without prefetcher does not suffer performance reduction. For class B, however, a small performance loss is observed even with all hardware prefetchers disabled. This points out that the source of performance loss for class A is strictly related to the previously noted contention that arises with the prefetcher action in a multi-core system. For class A, the working set appears to fit entirely within the on-core memory levels; consequently, the application only demonstrates performance loss when facing prefetcher-related contention. However, as the problem size scales, more off-core long-latency memory accesses are necessary since the working set does not fit entirely inside the on-core memory levels. For this reason, these long-latency memory accesses, together with the prefetcher-related memory contention, harm the application performance more fiercely in class B.

6 INVESTIGATING PREFETCHERS ON SIMULATION

As discussed in Chapter 1, parallel architecture simulation tools are necessary to develop and evaluate new techniques such as new prefetcher algorithms. A key requirement is that these simulation tools can effectively simulate the parallel architecture and the parallel applications, bringing similar values of the metrics when compared to the real execution. However, in Chapter 3 we have already pointed out several problems of working with computer architecture simulators. Specifically considering the prefetcher, the increasing variety of prefetching mechanisms and the trade-offs presented at Chapter 2.2 further complicate prefetch design in simulators. In this Chapter, we therefore aim to clarify the following question: *Using the ZSim and Sniper simulators, how do they behave and how accurately do they simulate NPB, accounting distinct prefetchers when possible?*

While simulators are a useful tool to test new architecture techniques, one drawback is the large amount of time that is often required by the simulations. For instance, the time to simulate the SP and BT applications of NPB using Sniper took more than one week. Sinuca (ALVES et al., 2015), which is another parallel architecture simulator, took even more time to simulate, exceeding the maximum execution time allowed by our computing infrastructure. In this regard, we had to discard Sinuca from the study and with Sniper we simulated only a subset of the NPB applications, notably CG, EP, FT, IS, LU, and MG. We also only analyzed the simulation of prefetchers with the Sniper simulator, since ZSim does not simulate memory prefetchers.

Figure 6.1 shows the obtained IPCs when no prefetcher is used for the simulations and the real execution of six NPB applications, namely CG, EP, FT, IS, MG, and LU, simulating the input class A. For that we disabled the prefetcher on both the real execution and the Sniper simulation. We did not make any specific changes to ZSim for this experiment, since ZSim does not support prefetcher simulation.

The only application where both simulators follow the real execution performance tendency is the EP. Since EP makes very little use of communication, it results in a simulation with very little contention events. This makes it easier for ZSim and Sniper to simulate EP, since simulating the effects of contention is arguably one of the most complex tasks in architecture simulation because of the out-of-order nature of contention events (SANCHEZ; KOZYRAKIS, 2013). However, for applications where communication and contention are more predominant, we can notice discrepancies between the

simulation and the real execution. This discrepancy can be quite extreme. For instance, the IS simulation with ZSim and with 12 threads resulted in an average IPC 30% lower than in the real execution, and Sniper CG simulation with 12 threads resulted in an average IPC 426% lower than in the real execution. The CG application is an interesting case because both Sniper and ZSim do not seem to accurately simulate the parallel CG execution (see Section 5.1). Both simulators predict a decrease in IPC as the number of threads increase, whereas in the real execution the trend is the opposite.

As aforementioned, accurately modeling contention is challenging. As mentioned in Section 4.1.1, in the ZSim case it is assumed that most concurrent accesses happen to unrelated cache lines when considering small time scales. ZSim was engineered in such a way that requests to the same cache line may be simulated in a different order than the observed in the real execution (SANCHEZ; KOZYRAKIS, 2013). Therefore, the path of the data through the cache hierarchy may change, resulting in simulation inaccuracies. This may explain the discrepancies of ZSim when simulating CG, as it is straightforward to devise that the concurrent and irregular memory accesses present in CG lead to concurrent accesses in same cache lines. This same explanation can also be the case for the LU application, though a more detailed memory access analysis of LU is required to attest this argument.

As a general trend, for NPB applications with communication and contention,

Figure 6.1: Obtained IPCs when no prefetcher is used for ZSim and Sniper simulations and the real execution.

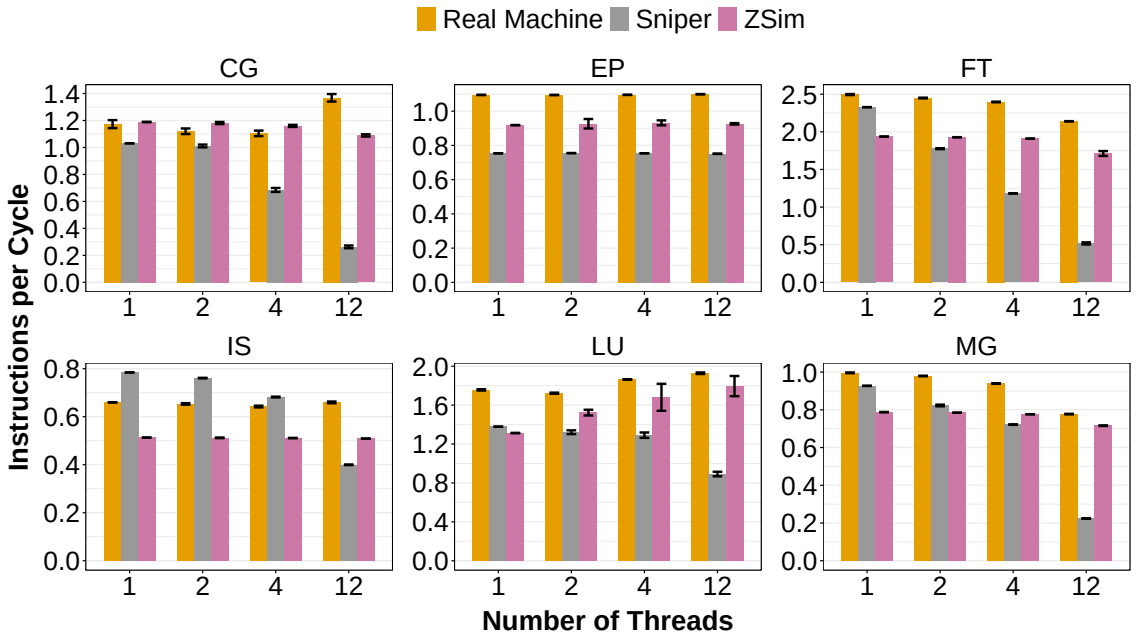
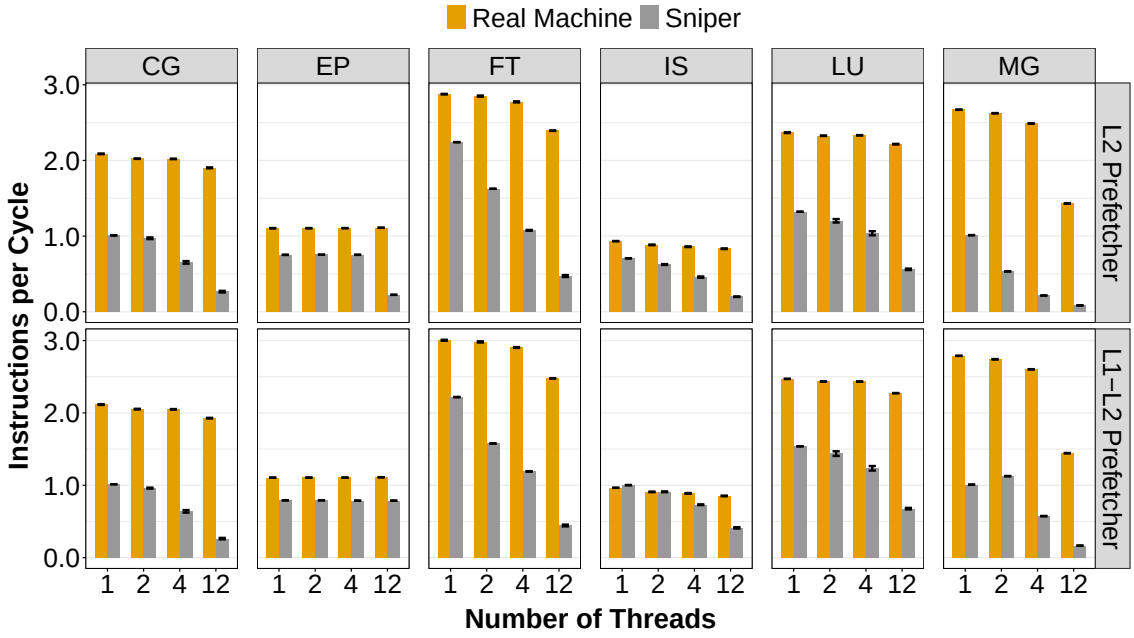


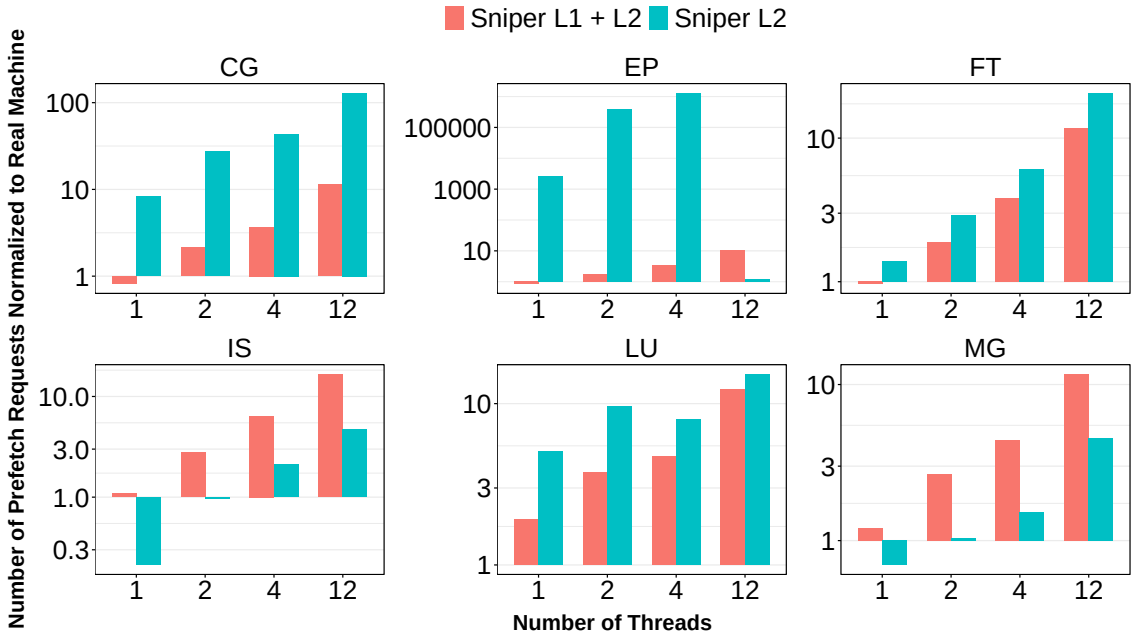
Figure 6.2: Comparison of Sniper L2 and L1+L2 prefetchers performance to the real executions results.



ZSim tends to underestimate the contention effects, while Sniper tends to overestimate the contention effects as the number of threads in the simulation increases. For ZSim, this fact can be due to the above-mentioned design assumptions made during its development. For Sniper, in its turn, the reason can be more complicated, as it has been reported (CARLSON et al., 2014) that simulation errors in the memory model of Sniper happen when simulating parallel applications with the interval model (see Section 4.1.2).

In Figure 6.2, we compare the performance, in terms of IPC, of the Sniper L2 prefetcher with the L2 prefetcher of the real machine, and the Sniper L1+L2 prefetcher with the real counterpart, respectively. In all evaluated NPB applications, we can notice discrepancies between the Sniper simulation results and the real machine results, with Sniper mostly underestimating the IPC performance, and overestimating the communication and contention effects with increasing number of threads. Many reasons can explain this fact. First, as mentioned above, Sniper presents errors in its memory model when simulating parallel applications, even without considering prefetcher. These errors may be further amplified by adding a memory prefetcher in the simulation. Second, Global History Buffer (GHB) (NESBIT; SMITH, 2004b) L2 prefetcher implemented in Sniper differs from the Stream (INTEL, 2019) L2 prefetcher present in Skylake. In this regard, the GHB prefetcher offered by Sniper may not be suited to simulate Skylake. Modeling prefetcher algorithms with the same level of specificity of the real algorithms may be unfeasible, unfortunately, since manufaturers need to conceal key characteristics of their

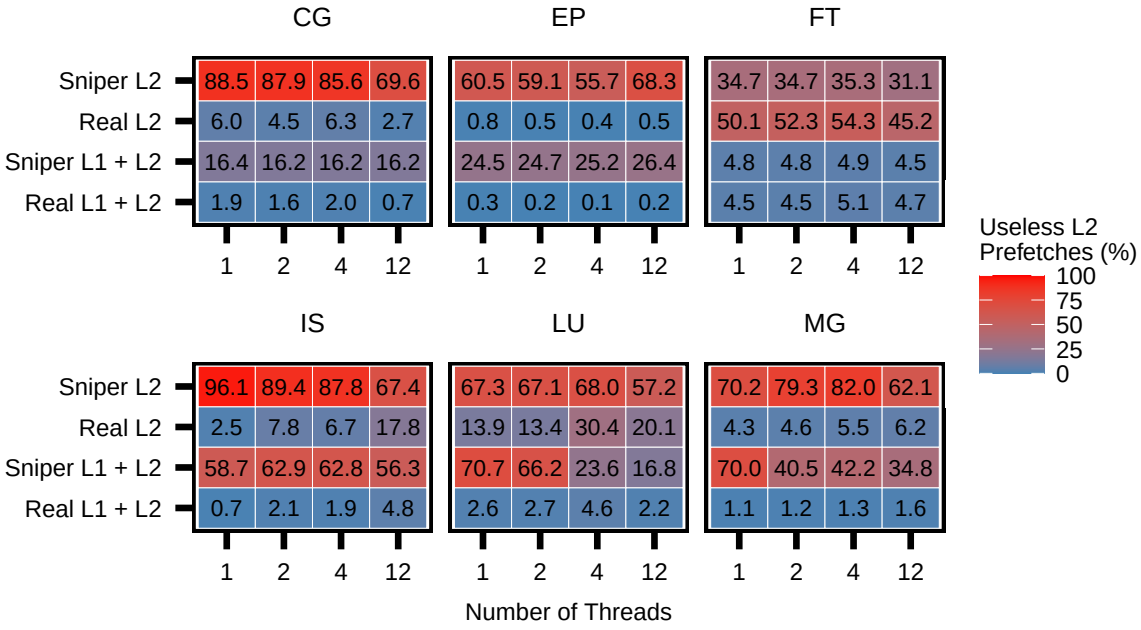
Figure 6.3: Number of prefetches issued by the simulation with the Sniper prefetchers, in ratio of their real counterparts. (log scale on y axis).



products (including prefetcher algorithms) in order to stay competitive. Another point worth noting is that, as mentioned in Section 4.2, Skylake implements a non-inclusive L3 cache. This non-inclusive L3 adds new complexity to the memory subsystem simulation since the data location in this new organization depends upon several aspects (INTEL, 2019). Similarly to other architecture simulators, Sniper implements an inclusive L3 cache. This difference can also contribute to the discrepancies between the real execution using Skylake and the Sniper simulation.

In Figure 6.3 we present the total number of prefetch requests performed by the Sniper L1+L2 prefetcher, the Sniper L2 prefetcher, in ratio of their real counterparts. The number of prefetch requests for the L1+L2 prefetcher counts the prefetches on both L1 and L2 caches. As expected at this point, Sniper L2 prefetcher model was not capable to accurately simulate the Skylake L2 prefetcher on the number of prefetch requests as well, presenting large discrepancies (notice the log scale on the y axis of Figure 6.3). Sniper L1+L2 prefetcher presented a total number of prefetch requests closer to the real execution. However, the similar number of prefetch requests of the Sniper L1+L2 prefetchers does not translate in similar estimations of the applications performance, as shown in Figure 6.2. This may be due to the fact that the prefetches performed by Sniper are different in terms of usefullness. To attest this argument, in Figure 6.4 we show the mean percentage of prefetches that were not useful during the executions in the real machine and in the

Figure 6.4: Useless prefeteches performed by the simulation and by the real hardware, in ratio of the total number of prefetches.



Sniper simulation. We can notice that the only application that sniper managed to accurately simulate the usefullness of the prefetches is the FT application. When simulating the other applications with Sniper, Sniper considered the majority of prefetch requests as useless, with useless prefetches close to 100% in some cases. This may indicate that Sniper memory simulation module is having issues on simulating how the prefetched data is interacting with cache hierarchy. The non-inclusive L3 cache may also be a cause of this issue, since again Sniper considers an inclusive L3 cache. These several results further emphasize that simulators are often designed for specific microarchitectures and can not be used as general tools (NOWATZKI et al., 2015).

7 DISCUSSION AND FINDINGS

An important aspect highlighted throughout the entire work is the difficulty of working with the memory system, with the prefetcher and its algorithms being a special case. The obscurity and confidentiality around the real implementation makes accurate models and algorithms impossible to be reproduced in simulators; moreover, for the same reasons above, simulator users have difficulties in finding the proper parameters for the prefetcher models. With the simulation design being focused on specific microarchitectures and its details, the employment of simulators on memory-related research can be easily questioned (DESIKAN; BURGER; KECKLER, 2001; NOWATZKI et al., 2015).

As proposed on the Section 2.2, there are several trade-offs involved on the prefetcher design that directly impact the performance. For instance, overly aggressive prefetchers are likely to generate pollution and thrashing, which exacerbate contention and power consumption, and inter-core interference was broadly observed throughout this entire work. In our experiments, we found interesting characteristics in the studied prefetchers that may direct future works. We have discussed how the L1 data prefetcher snooping the L1 cache requests can create contention by occupying the limited line fill buffers in the L1 cache. Considering this and the L2 prefetcher impact on performance, prefetching algorithms that consider their interaction and their peculiarities may be beneficial to performance. Moreover, the interaction between the L1 prefetcher and the L2 prefetcher makes architecture studies that consider prefetchers harder to interpret. Analyzing applications' performance, therefore, becomes harder due to the lack of control over the prefetchers and their implementations.

We argue that the use of both prefetchers (L1+L2) does not necessarily warrant significant performance gains, which is not intuitive. When considering the L2 prefetcher, we obtained performance gains similar to when using both prefetchers, with the advantages of having more control over the experiments and the noise in the memory subsystem, faster simulation time, and less energy consumption due to the smaller number of prefetch requests being performed. This is comprehensible, as the current cache memory technologies allow a small difference in latency between the L1 and L2 caches, while the access latency to the off-chip L3 cache memory is considerably higher than the access latency of the L2 cache (on average 7x, due to the mesh topology necessary to fully utilize the cache, as it is distributed in L3 off-core banks). Therefore, avoiding accesses to the L3 cache is more critical in terms of latency, and it is advisable to use the L2 prefetcher as

a standalone instead of both prefetchers (which is the default setting). The L2 prefetcher observes the more relevant access patterns, as it prefetches data that would always require an access to the L3, thus hiding the large L3 latency.

With the increase in application parallelism, the execution time naturally becomes smaller. However, we see that the performance per core decreases as we increase the level of parallelism, showing the impact of memory accesses and communication over the application performance. As the amount of communication increases, the contention for the shared resources (interconnection network and LLC banks) increases as well. Thereby, the request buffers of the caches and main memory become full, and the prefetcher cannot sufficiently mitigate the memory latency, resulting in small IPC values. Thus, analyzing the communication pattern of applications is an important task to improve their performance, as applications which exert heavy memory pressure or communicate often can diminish the usefulness of a prefetcher.

The degradation of the prefetcher usefulness has already been observed in other applications, such as the experiences reported by Dell (DELL, 2006), which claim 8% increased performance for the SAP NetWeaver Portal application when disabling the hardware prefetchers. In recent years, Intel implemented forms of reducing the prefetcher aggressiveness due to multiple reports such as Dell's, in order to avoid performance degradation. Our real execution results show that even with high parallelism, Intel's strategy works, as the prefetcher never loses performance compared to an execution without prefetchers. However, it would be interesting to test this assumption with simultaneous multi-threading.

Nevertheless, our research shows that computer architecture researchers should always implement a prefetcher aggressiveness attenuator in the simulator model, and test their new implementations on highly parallel applications which exert memory pressure and inter-core interference (EBRAHIMI et al., 2009), so that the possible negative impacts of their design can be properly evaluated.

The main scientific and technical findings that we experienced during the development of this work are summarized below. These findings are framed into a list of good practices and guidelines that we consider helpful and advisable for future research and development on memory prefetcher with parallel applications, with and without simulation.

- It is recommended to pay attention on how the profiling tool is collecting performance data. Some profiling tools (such as Perf (MELO, 2010) and PAPI (TERP-

STRA et al., 2010)) collect performance data in either a system-wide (Perf) or application-wide (Perf and PAPI) manner. For the former, system processes may affect the profiling results. Therefore, when evaluating the performance of applications individually, we must assure that the profiling tool is collecting performance data application-wide;

- One should be careful if a certain architecture research (notably prefetcher research) relies on an evaluation/validation performed by architecture simulators. In the acquired experience, we noticed that architecture simulators did not accurately represent real architectures, specifically when we consider memory prefetcher and parallel applications. Improving simulators' accuracy is a concerning and essential matter for the future of architecture research;
- Research on prefetcher algorithms for the L2 cache seems more promising to yield significant performance improvements – as opposed to prefetcher algorithms for the L1 cache – since, in our experience, accessing the L3 cache was a larger detriment to performance, as opposed to accessing the L2 cache;
- For highly parallel applications (in the order of dozens of threads on a single machine), memory prefetchers may not be critical for the application performance since, in our experience, the prefetchers' performance seems to be bound by the contention for memory access. When contention is created due to the large number of requests that arrive on shared resources, prefetch requests might get so late that they cannot hide the memory latency.

8 CONCLUSION AND FUTURE WORK

Prefetching algorithms have been widely used in processors to mitigate the performance gap between the processors and the memory subsystem. Analyzing and developing new prefetching algorithms is a notoriously challenging task, especially due to the complexities and obscurities behind computer architecture development. Hardware prefetching research is mainly possible thanks to architecture simulators that attempt to model the highly complex (and sometimes obscure) interactions present in the hardware. When we account for parallel, High-Performance Computing (HPC) applications, understanding the prefetcher contribution to performance, on both the real hardware and in the simulations, becomes an important matter.

In this work we performed an experimental investigation of the prefetcher role in the performance of parallel applications. Our investigation included a pioneer study on the prefetcher performance in a simulated environment, taking the Sniper simulator as an example. Several insights were obtained regarding methodological aspects, the behavior of the studied prefetcher models, and how researchers and end users should handle prefetchers, in a both real and simulated scenario.

Among these insights, one can highlight: (i) prefetching from the L3 to L2 cache presents a more substantial contribution to performance, (ii) memory contention becomes a larger constraint in the performance as the level of parallelism increases, narrowing the prefetcher contribution, (iii) Skylake parallel memory contention is poorly simulated by ZSim and Sniper, and (iv) the non-inclusive L3 cache present in the Skylake architecture hinders the accurate simulation of NPB with the Sniper prefetchers.

Since we observed that some prefetcher combinations perform unnecessary costly DRAM accesses, it is perhaps relevant to evaluate how these inaccurate prefetch requests impact energy consumption. Therefore, for future work, we intend to analyze the prefetcher impact on energy consumption and Energy-Delay Product (EDP) (GONZALEZ; HOROWITZ, 1996), since, for instance, the EDP may help us understand which prefetcher configurations are more suitable considering both energy efficiency and execution time.

8.1 Published Papers

Several works assisted on the development of this monograph. The following set represents publications that are directly related to the contributions presented in this work:

- **Valéria S. Girelli**, Francis B. Moreira, Matheus S. Serpa, and Philippe O. A. Navaux. *"Impacto do Prefetcher na Precisão de Simulações de Arquiteturas Paralelas"*. In XX Symposium on High-Performance Computing Systems (WSCAD), **2019**. The full paper was published at the main track and received a Mention of Honor Award.
- **Valéria S. Girelli**, Francis B. Moreira, Matheus S. Serpa, Danilo Carastan-Santos, and Philippe O. A. Navaux. *"Investigating memory prefetcher performance over parallel applications: From real to simulated"*. Concurrency Computation: Practice and Experience, **2021. Qualis A2**.

This work was also presented at the following workshop:

- **6th Career Workshop for Women and Minorities in Computer Architecture (CWWMCA 2020):** workshop in conjunction with the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO-53). *"Assessing the Prefetcher's Role in High-Performance Computing"*. **Valéria S. Girelli**, Francis B. Moreira, Matheus S. Serpa, Danilo Carastan-Santos, and Philippe O. A. Navaux.

Besides the directly related publications, the following collaboration is also related to this work:

- Arthur Krause, Francis B. Moreira, **Valéria S. Girelli**, and Philippe O. A. Navaux. *"Poluição de Cache e Thrashing em Aplicações Paralelas de Alto Desempenho"*. In XX Symposium on High-Performance Computing Systems (WSCAD), **2019**.

REFERENCES

AARNO, D.; ENGBLOM, J. **Software and System Development using Virtual Platforms - Full-System Simulation with Wind River Simics**. [S.l.: s.n.], 2014. ISBN 9780128007259.

ADILEH, A. et al. Racing to hardware-validated simulation. In: WENISCH, T.; AGARWAL, N. (Ed.). **2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2019. p. 58–67.

AHN, J. H. et al. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In: . [S.l.: s.n.], 2013. p. 74–85. ISBN 978-1-4673-5776-0.

AKRAM, A.; SAWALHA, L. A survey of computer architecture simulation techniques and tools. **IEEE Access**, IEEE, 2019.

ALVES, M. A. Z. et al. Sinuca: A validated micro-architecture simulator. In: QIU, M. (Ed.). **2015 IEEE 17th International Conference on High Performance Computing and Communications**. [S.l.], 2015. p. 605–610.

AUSTIN, T.; LARSON, E.; ERNST, D. Simplescalar: An infrastructure for computer system modeling. **Computer**, IEEE, n. 2, p. 59–67, 2002.

BAER, J.-L.; CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In: **Proceedings of the 1991 ACM/IEEE Conference on Supercomputing**. New York, NY, USA: Association for Computing Machinery, 1991. (Supercomputing '91), p. 176–186. ISBN 0897914597. Available from Internet: <https://doi.org/10.1145/125826.125932>.

BAKHSHALIPOUR, M.; LOTFI-KAMRAN, P.; SARBAZI-AZAD, H. Domino temporal data prefetcher. In: **2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2018. p. 131–142.

BAKHSHALIPOUR, M. et al. Bingo spatial data prefetcher. In: LOURI, A.; VENKATARAMANI, G.; GRATZ, P. (Ed.). **2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.], 2019. p. 399–411.

BAKHSHALIPOUR, M. et al. Evaluation of hardware data prefetchers on server processors. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 3, jun. 2019. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3312740>.

BHATIA, E. et al. Perceptron-based prefetch filtering. In: **Proceedings of the 46th International Symposium on Computer Architecture**. New York, NY, USA: Association for Computing Machinery, 2019. (ISCA '19), p. 1–13. ISBN 9781450366694. Available from Internet: <https://doi.org/10.1145/3307650.3322207>.

BIENIA, C. et al. The parsec benchmark suite: Characterization and architectural implications. In: MOSHOVOS, A.; TARDITI, D.; OLUKOTUN, K. (Ed.). **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. [S.l.], 2008. p. 72–81.

BINKERT, N. et al. The gem5 simulator. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 2, p. 1–7, aug. 2011. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/2024716.2024718>.

CANTIN, J.; LIPASTI, M.; SMITH, J. Stealth prefetching. In: . [S.l.: s.n.], 2006. v. 40, p. 274–282.

CARLSON, T. E. et al. An evaluation of high-level mechanistic core models. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, New York, NY, USA, 2014. ISSN 1544-3566.

CARLSON, T. E.; HEIRMANT, W.; EECKHOUT, L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: LATHROP, S.; COSTA, J.; KRAMER, W. (Ed.). **SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2011. p. 1–12.

CHARNEY, M. J.; REEVES, A. P. **Generalized correlation based hardware prefetching**. 1995. <https://www.semanticscholar.org/paper/Generalized-correlation-based-hardware-prefetching-Charney-Reeves/86b3c8787a63f4d1d30ff53ac115cbd8881a7af7>. [Accesed in: 18 Apr. 2020].

CHEN, C. F. et al. Accurate and complexity-effective spatial pattern prediction. In: **10th International Symposium on High Performance Computer Architecture (HPCA'04)**. [S.l.: s.n.], 2004. p. 276–287.

CHEN, T.-F.; BAER, J.-L. Effective hardware-based data prefetching for high-performance processors. **IEEE transactions on computers**, IEEE, v. 44, n. 5, p. 609–623, 1995.

CHOU, Y. Low-cost epoch-based correlation prefetching for commercial applications. In: **40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)**. [S.l.: s.n.], 2007. p. 301–313.

CMELIK, B.; KEPPEL, D. Shade: A fast instruction-set simulator for execution profiling. **SIGMETRICS Perform. Eval. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 1, p. 128–137, may 1994. ISSN 0163-5999. Available from Internet: <https://doi.org/10.1145/183019.183032>.

CRUZ, E. H.; DIENER, M.; NAVAUX, P. O. **Thread and Data Mapping for Multicore Systems: Improving Communication and Memory Accesses**. [S.l.]: Springer, 2018.

CUTRESS, I. **The AMD Zen and Ryzen 7 Review: A Deep Dive on 1800X, 1700X and 1700**. 2017. Available from Internet: <https://www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700>.

DELL. **Dell SAP NetWeaver Benchmark Info**. 2006. <https://www.dell.com/downloads/global/solutions/Dell_SAP_NetWeaver_Benchmark_Info.pdf>. [Accesed in: 24 Mar. 2020].

DESIKAN, R.; BURGER, D.; KECKLER, S. Measuring experimental error in microprocessor simulation. **ACM SIGSOFT Software Engineering Notes**, v. 26, p. 266–277, 05 2001.

DOWECK, J. et al. Inside 6th-generation intel core: New microarchitecture code-named skylake. **IEEE Micro**, IEEE Computer Society Press, Washington, DC, USA, v. 37, n. 2, p. 52–62, mar. 2017. ISSN 0272-1732. Available from Internet: <https://doi.org/10.1109/MM.2017.38>.

DUBOIS, M.; ANNAVARAM, M.; STENSTRM, P. **Parallel Computer Organization and Design**. USA: Cambridge University Press, 2012. ISBN 0521886759.

EBRAHIMI, E. et al. Coordinated control of multiple prefetchers in multi-core systems. In: ALBONESI, D. et al. (Ed.). **Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2009. p. 316–326.

EECKHOUT, L. Computer architecture performance evaluation methods. In: **Computer Architecture Performance Evaluation Methods**. [S.l.: s.n.], 2010.

ELRABAA, M. E. S. et al. A very fast trace-driven simulation platform for chip-multiprocessors architectural explorations. **IEEE Transactions on Parallel and Distributed Systems**, v. 28, n. 11, p. 3033–3045, 2017.

ESMAILI-DOKHT, P. et al. **Scale-Out Processors Energy Efficiency**. 2018.

EYERMAN, S. et al. A mechanistic performance model for superscalar out-of-order processors. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 27, n. 2, may 2009. ISSN 0734-2071. Available from Internet: <https://doi.org/10.1145/1534909.1534910>.

FERDMAN, M. et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 47, n. 4, p. 37–48, mar. 2012. ISSN 0362-1340. Available from Internet: <https://doi.org/10.1145/2248487.2150982>.

FERDMAN, M.; FALSAFI, B. Last-touch correlated data streaming. In: **2007 IEEE International Symposium on Performance Analysis of Systems Software**. [S.l.: s.n.], 2007. p. 105–115.

FOG, A. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. **Copenhagen University College of Engineering**, p. 02–29, 2012.

GENBRUGGE, D.; EYERMAN, S.; EECKHOUT, L. Interval simulation: Raising the level of abstraction in architectural simulation. In: DAS, C.; CJACOB, M.; YANG, Q. (Ed.). **Proceedings - International Symposium on High-Performance Computer Architecture**. [S.l.], 2010. p. 1 – 12.

GIRELLI, V. S. et al. Investigating memory prefetcher performance over parallel applications: From real to simulated. **Concurrency and Computation: Practice and Experience**, n/a, n. n/a, p. e6207. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6207>.

GONZALEZ, R.; HOROWITZ, M. Energy dissipation in general purpose microprocessors. **IEEE Journal of Solid-State Circuits**, v. 31, n. 9, p. 1277–1284, 1996.

GUTIERREZ, A. et al. Sources of error in full-system simulation. In: AAMODT, T.; HEMPSTEAD, M. (Ed.). **2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2014. p. 13–22.

GUTTMAN, D. et al. Performance and energy evaluation of data prefetching on intel xeon phi. In: **2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2015. p. 288–297.

HAMMARLUND, P. et al. Haswell: The fourth-generation intel core processor. **IEEE Micro**, IEEE, v. 34, n. 2, p. 6–20, 2014.

HARDAVELLAS, N. et al. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. **SIGMETRICS Perform. Eval. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 4, p. 31–34, mar. 2004. ISSN 0163-5999. Available from Internet: <https://doi.org/10.1145/1054907.1054914>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Sixth Edition: A Quantitative Approach**. 6th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055.

HUGHES, C. J. et al. Rsim: simulating shared-memory multiprocessors with ilp processors. **Computer**, v. 35, n. 2, p. 40–49, 2002.

IACOBOVICI, S. et al. Effective stream-based and execution-based data prefetching. In: **Proceedings of the 18th Annual International Conference on Supercomputing**. New York, NY, USA: Association for Computing Machinery, 2004. (ICS '04), p. 1–11. ISBN 1581138393. Available from Internet: <https://doi.org/10.1145/1006209.1006211>.

INTEL. **Intel® 64 and IA-32 Architectures Optimization Reference Manual**. 2019. <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>. [Accesed in: 16 Jan. 2020].

INTEL. **Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers**. 2021. <https://software.intel.com/content/dam/develop/external/us/en/documents-tps/335592-sdm-vol-4.pdf>. [Accesed in: 18 Jan. 2020].

ISHII, Y.; INABA, M.; HIRAKI, K. Access map pattern matching for data cache prefetch. In: . [S.l.: s.n.], 2009. p. 499–500.

JAIN, A.; LIN, C. Linearizing irregular memory accesses for improved correlated prefetching. In: **Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture**. New York, NY, USA: Association for Computing Machinery, 2013. (MICRO-46), p. 247–259. ISBN 9781450326384. Available from Internet: <https://doi.org/10.1145/2540708.2540730>.

JAIN, A.; LIN, C. Rethinking belady's algorithm to accommodate prefetching. In: BILOF, R. (Ed.). **2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.], 2018. p. 110–123.

JALEEL, A. et al. Cmp$im: A pin-based on-the-fly multi-core cache simulator. **Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)**, p. 28–36, 01 2008.

JALEEL, A. et al. High performance cache replacement using re-reference interval prediction (rrip). In: **Proceedings of the 37th Annual International Symposium on Computer Architecture**. New York, NY, USA: Association for Computing Machinery, 2010. (ISCA '10), p. 60–71. ISBN 9781450300537. Available from Internet: <https://doi.org/10.1145/1815961.1815971>.

JIN, H. et al. **The OpenMP Implementation of NAS Parallel Benchmarks and its Performance**. NASA Ames Research Center, 1999.

JOUPPI, N. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In: . [S.l.: s.n.], 1998. v. 18, p. 388–397.

KANG, H.; WONG, J. L. To hardware prefetch or not to prefetch? a virtualized environment study and core binding approach. In: **Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: Association for Computing Machinery, 2013. (ASPLOS '13), p. 357–368. ISBN 9781450318709. Available from Internet: <https://doi.org/10.1145/2451116.2451155>.

KAYNAK, C.; GROT, B.; FALSAFI, B. Shift: Shared history instruction fetch for lean-core server processors. In: **2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2013. p. 272–283.

KIM, J. et al. Path confidence based lookahead prefetching. In: **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2016. p. 1–12.

KIM, T.; ZHAO, D.; VEIDENBAUM, A. V. Multiple stream tracker: A new hardware stride prefetcher. In: **Proceedings of the 11th ACM Conference on Computing Frontiers**. New York, NY, USA: Association for Computing Machinery, 2014. (CF '14). ISBN 9781450328708. Available from Internet: <https://doi.org/10.1145/2597917.2597941>.

KISE, K. et al. The simcore/alpha functional simulator. In: **WCAE '04**. [S.l.: s.n.], 2004.

Kumar, S.; Wilkerson, C. Exploiting spatial locality in data caches using spatial footprints. In: **Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)**. [S.l.: s.n.], 1998. p. 357–368.

LE, H. Q. et al. Ibm power6 microarchitecture. **IBM Journal of Research and Development**, IBM, v. 51, n. 6, p. 639–662, 2007.

LEE, C. J. et al. Prefetch-aware dram controllers. In: GONZALEZ, A. et al. (Ed.). **Proceedings of the 41st Annual IEEE/ACM international Symposium on Microarchitecture**. [S.l.], 2008. p. 200–209.

LOTFI-KAMRAN, P. et al. Scale-out processors. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 40,

n. 3, p. 500–511, jun. 2012. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/2366231.2337217>.

MARTIN, M. M. K. et al. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 4, p. 92–99, nov. 2005. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/1105734.1105747>.

MELO, A. C. de. The New Linux 'perf' Tools. In: KONGRESS, L. (Ed.). **International Linux System Technology Conference**. [S.l.], 2010. p. 1–42.

MICHAUD, P. Best-offset hardware prefetching. **2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)**, p. 469–480, 2016.

MILLER, J. E. et al. Graphite: A distributed parallel simulator for multicores. In: **HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture**. [S.l.: s.n.], 2010. p. 1–12.

MITTAL, S. A survey of recent prefetching techniques for processor caches. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 49, n. 2, aug. 2016. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/2907071>.

MORGAN, T. P. **Drilling Xeon Skylake Architecture**. 2017. Available from Internet: <https://www.nextplatform.com/2017/08/04/drilling-xeon-skylake-architecture/>.

NESBIT, K. J.; DHODAPKAR, A. S.; SMITH, J. E. Ac/dc: an adaptive data cache prefetcher. In: **Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.** [S.l.: s.n.], 2004. p. 135–145.

NESBIT, K. J.; SMITH, J. E. Data cache prefetching using a global history buffer. In: **10th International Symposium on High Performance Computer Architecture (HPCA'04)**. [S.l.: s.n.], 2004. p. 96–96.

NESBIT, K. J.; SMITH, J. E. Data cache prefetching using a global history buffer. In: TIRADO, F.; ZAPATA, E. (Ed.). **10th International Symposium on High Performance Computer Architecture (HPCA'04)**. [S.l.], 2004. p. 96–96.

NORI, A. V. et al. Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies. In: **Proceedings of the 45th Annual International Symposium on Computer Architecture**. IEEE Press, 2018. (ISCA '18), p. 96–109. ISBN 9781538659847. Available from Internet: <https://doi.org/10.1109/ISCA.2018.00019>.

NOWATZKI, T. et al. Architectural simulators considered harmful. **IEEE Micro**, v. 35, n. 6, p. 4–12, 2015.

PATEL, A.; AFRAM, F.; GHOSE, K. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In: MUELLER, W.; PADERBORN, U.; PETROT, F. (Ed.). **1st International Qemu Users' Forum**. [S.l.], 2011. p. 29–30.

PUGSLEY, S. H. et al. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In: **2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2014. p. 626–637.

REDDI, V. J. et al. Pin: A binary instrumentation tool for computer architecture research and education. In: **Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture**. New York, NY, USA: Association for Computing Machinery, 2004. (WCAE '04), p. 22–es. ISBN 9781450347334. Available from Internet: <https://doi.org/10.1145/1275571.1275600>.

RENAU, J. et al. **SESC simulator**. 2005. Http://sesc.sourceforge.net.

ROTEM, E. et al. Power-management architecture of the intel microarchitecture code-named sandy bridge. **IEEE Micro**, v. 32, n. 2, p. 20–27, 2012.

SANCHEZ, D. **ZSim Tutorial Validation**. 2016. <http://zsim.csail.mit.edu/tutorial/slides/validation.pdf>. [Accesed in: 10 Sept. 2019].

SANCHEZ, D.; KOZYRAKIS, C. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 41, n. 3, p. 475–486, jun. 2013. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/2508148.2485963>.

SANGAIAH, K. et al. Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of cmp and hpc workloads. **ACM Trans. Archit. Code Optim.**, v. 15, p. 2:1–2:26, 2018.

SHARKEY, J.; PONOMAREV, D.; GHOSE, K. Abstract m-sim: A flexible, multithreaded architectural simulation environment. 01 2005.

SHEVGOOR, M. et al. Efficiently prefetching complex address patterns. In: **2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2015. p. 141–152.

SKADRON, K. et al. Challenges in computer architecture evaluation. **Computer**, IEEE, v. 36, n. 8, p. 30–36, 2003.

SOLIHIN, Y.; LEE, J.; TORRELLAS, J. Using a user-level memory thread for correlation prefetching. In: **Proceedings 29th Annual International Symposium on Computer Architecture**. [S.l.: s.n.], 2002. p. 171–182.

SOMOGYI, S. et al. Spatial memory streaming. In: **33rd International Symposium on Computer Architecture (ISCA'06)**. [S.l.: s.n.], 2006. p. 252–263.

SRINATH, S. et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In: LOURI, A.; YOUSIF, M.; BYRD, G. (Ed.). **2007 IEEE 13th International Symposium on High Performance Computer Architecture**. [S.l.], 2007. p. 63–74.

TERPSTRA, D. et al. Collecting performance data with papi-c. In: . [S.l.]: Tools for High Performance Computing. Springer, 2010. p. 157–173.

UBAL, R. et al. Multi2sim: a simulation framework for cpu-gpu computing. In: YEW, P.-C. et al. (Ed.). **2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)**. [S.l.], 2012. p. 335–344.

VEENSTRA, J. E.; FOWLER, R. J. Mint: a front end for efficient simulation of shared-memory multiprocessors. In: **Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems**. [S.l.: s.n.], 1994. p. 201–207.

WALKER, M. et al. Hardware-validated cpu performance and energy modelling. In: NIKOLOPOULOS, D.; SUPINSKI, B.; DELIMITROU, C. (Ed.). **2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2018. p. 44–53.

WENISCH, T. F. et al. Practical off-chip meta-data for temporal memory streaming. In: **2009 IEEE 15th International Symposium on High Performance Computer Architecture**. [S.l.: s.n.], 2009. p. 79–90.

WENISCH, T. F. et al. Temporal streaming of shared memory. In: **Proceedings of the 32nd Annual International Symposium on Computer Architecture**. USA: IEEE Computer Society, 2005. (ISCA '05), p. 222–233. ISBN 076952270X. Available from Internet: <https://doi.org/10.1109/ISCA.2005.50>.

WOO, S. C. et al. The splash-2 programs: Characterization and methodological considerations. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 23, n. 2, p. 24–36, may 1995. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/225830.223990>.

WU, C. et al. Pacman: Prefetch-aware cache management for high performance caching. In: **2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2011. p. 442–453.

WU, H. et al. Efficient metadata management for irregular data prefetching. In: MANNE, S.; HUNTER, H.; ALTMAN, E. (Ed.). **Proceedings of the 46th International Symposium on Computer Architecture**. [S.l.], 2019. p. 449–461.

WULF, W.; MCKEE, S. Hitting the memory wall: Implications of the obvious. **Computer Architecture News**, v. 23, 01 1996.

YOURST, M. T. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In: ALBONESI, D.; BROOKS, D. (Ed.). **2007 IEEE International Symposium on Performance Analysis of Systems & Software**. [S.l.], 2007. p. 23–34.

ZHU, H.; CHEN, Y.; SUN, X.-H. Timing local streams: Improving timeliness in data prefetching. In: **Proceedings of the 24th ACM International Conference on Supercomputing**. New York, NY, USA: Association for Computing Machinery, 2010. (ICS '10), p. 169–178. ISBN 9781450300186. Available from Internet: <https://doi.org/10.1145/1810085.1810110>.

ZHUANG, X.; HSIEN-HSIN, S. L. Reducing cache pollution via dynamic data prefetch filtering. **IEEE Transactions on Computers**, IEEE, v. 56, n. 1, p. 18–31, 2006.